

RP2040 Datasheet

A microcontroller
by Raspberry Pi

RP2040 Datasheet

A microcontroller
by Raspberry Pi

Colophon

© 2020-2025 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.)

This documentation is licensed under a Creative Commons [Attribution-NoDerivatives 4.0 International \(CC BY-ND\)](#).

Portions Copyright © 2019 Synopsys, Inc.

All rights reserved. Used with permission. Synopsys & DesignWare are registered trademarks of Synopsys, Inc.

Portions Copyright © 2000-2001, 2005, 2007, 2009, 2011-2012, 2016 Arm Limited.

All rights reserved. Used with permission.

build-date: 2025-02-20

build-version: 3184e62-clean

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL's [Standard Terms](#). RPL's provision of the RESOURCES does not expand or otherwise modify RPL's [Standard Terms](#) including but not limited to the disclaimers and warranties expressed in them.

版权页

© 2020-2025 Raspberry Pi Ltd (前称 Raspberry Pi (Trading) Ltd.)

本文件依据知识共享署名-禁止演绎 4.0 国际许可协议 (CC BY-ND) 授权。

部分版权归 Synopsys, Inc. © 2019 所有

版权所有。经许可使用。Synopsys 与 DesignWare 均为 Synopsys, Inc. 注册商标。部分版权归 Arm Limited © 2000-2001、2005、2007、2009、2011-2012、2016 所有。

版权所有。经许可使用。

构建日期: 2025-02-2

0 构建版本: 3184e62-clean

法律免责声明通知

Raspberry Pi 产品（包括数据表）不时修订的技术和可靠性数据（以下简称“资源”）由 Raspberry Pi Ltd（以下简称“RPL”）按“现状”提供，并否认任何明示或暗示的保证，包括但不限于对适销性及特定用途适用性的暗示保证。在适用法律允许的最大范围内，无论基于合同、严格责任或侵权（包括过失或其他）理论，RPL 对因使用该资源所引起的任何直接、间接、附带、特殊、惩罚性或后续性损害（包括但不限于采购替代商品或服务；使用权丧失、数据或利润损失；或业务中断）均不承担任何责任，即便已被告知发生此类损害的可能性。

RPL保留随时对资源（RESOURCES）或其中所描述的任何产品进行任何增强、改进、更正或其他修改的权利，且无需另行通知。

该资源（RESOURCES）仅供具备相应设计知识水平的熟练用户使用。用户应自行承担对资源（RESOURCES）的选择和使用以及对其中所描述产品的任何应用所产生的全部责任。用户同意赔偿并使RPL免受因其使用资源（RESOURCES）而产生的所有责任、费用、损害或其他损失的影响。

RPL授权用户仅可将资源（RESOURCES）与Raspberry Pi产品结合使用。禁止对资源（RESOURCES）作任何其他用途。未授予RPL或任何第三方的其他知识产权许可。

高风险活动。Raspberry Pi产品并非为在要求故障安全性能的危险环境中使用而设计、制造或意图使用，如核设施运行、航空导航或通信系统、空中交通管制、武器系统或安全关键应用（包括生命维持系统及其他医疗设备），在这些环境中产品的故障可能直接导致死亡、人身伤害或严重的身体或环境损害（以下简称“高风险活动”）。RPL明确否认对用于高风险活动的适用性的任何明示或暗示保证，且对将Raspberry Pi产品用于高风险活动或包含于其中不承担任何责任。

Raspberry Pi 产品依据 RPL 标准条款提供。RPL 提供的资源（RESOURCES）不构成对 RPL 标准条款的扩展或修改，包括但不限于其中明确表达的免责声明及保证条款。

Table of contents

Colophon	1
Legal disclaimer notice	1
1. Introduction	8
1.1. Why is the chip called RP2040?	8
1.2. Summary	9
1.3. The Chip	9
1.4. Pinout Reference	10
1.4.1. Pin Locations	10
1.4.2. Pin Descriptions	11
1.4.3. GPIO Functions	12
2. System Description	14
2.1. Bus Fabric	14
2.1.1. AHB-Lite Crossbar	15
2.1.2. Atomic Register Access	17
2.1.3. APB Bridge	17
2.1.4. Narrow IO Register Writes	17
2.1.5. List of Registers	18
2.2. Address Map	24
2.2.1. Summary	24
2.2.2. Detail	24
2.3. Processor subsystem	26
2.3.1. SIO	27
2.3.2. Interrupts	60
2.3.3. Event Signals	61
2.3.4. Debug	61
2.4. Cortex-M0+	62
2.4.1. Features	62
2.4.2. Functional Description	64
2.4.3. Programmer's model	68
2.4.4. System control	73
2.4.5. NVIC	74
2.4.6. MPU	76
2.4.7. Debug	76
2.4.8. List of Registers	77
2.5. DMA	91
2.5.1. Configuring Channels	91
2.5.2. Starting Channels	93
2.5.3. Data Request (DREQ)	95
2.5.4. Interrupts	96
2.5.5. Additional Features	96
2.5.6. Example Use Cases	97
2.5.7. List of Registers	101
2.6. Memory	120
2.6.1. ROM	120
2.6.2. SRAM	121
2.6.3. Flash	122
2.7. Boot Sequence	128
2.8. Bootrom	128
2.8.1. Processor Controlled Boot Sequence	129
2.8.2. Launching Code On Processor Core 1	131
2.8.3. Bootrom Contents	132
2.8.4. USB Mass Storage Interface	143
2.8.5. USB PICOBLOCK Interface	144
2.9. Power Supplies	150
2.9.1. Digital IO Supply (IOVDD)	151

目录

版权页	1
法律免责声明通知	1
1. 引言	8
1.1. 芯片为何命名为 RP2040?	8
1.2. 概要	9
1.3. 芯片	9
1.4. 引脚参考	10
1.4.1. 引脚位置	10
1.4.2. 引脚描述	11
1.4.3. GPIO 功能	12
2. 系统描述	14
2.1. 总线结构	14
2.1.1. AHB-Lite 交叉开关	15
2.1.2. 原子寄存器访问	17
2.1.3. APB 桥接	17
2.1.4. 窄带 IO 寄存器写入	17
2.1.5. 寄存器列表	18
2.2. 地址映射	24
2.2.1. 概述	24
2.2.2. 详细说明	24
2.3. 处理器子系统	26
2.3.1. SIO	27
2.3.2. 中断	60
2.3.3. 事件信号	61
2.3.4. 调试	61
2.4. Cortex-M0+	62
2.4.1. 功能特性	62
2.4.2. 功能描述	64
2.4.3. 程序员模型	68
2.4.4. 系统控制	73
2.4.5. NVIC	74
2.4.6. MPU	76
2.4.7. 调试	76
2.4.8. 寄存器列表	77
2.5. DMA	91
2.5.1. 通道配置	91
2.5.2. 启动通道	93
2.5.3. 数据请求 (DREQ)	95
2.5.4. 中断	96
2.5.5. 附加功能	96
2.5.6. 示例用例	97
2.5.7. 寄存器列表	101
2.6. 内存	120
2.6.1. 只读存储器 (ROM)	120
2.6.2. 静态随机存取存储器 (SRAM)	121
2.6.3. 闪存	122
2.7. 启动顺序	128
2.8. 启动只读存储器 (Bootrom)	128
2.8.1. 处理器控制的启动顺序	129
2.8.2. 在处理器核1上启动代码	131
2.8.3. 启动只读存储器内容	132
2.8.4. USB大容量存储设备接口	143
2.8.5. USB PICOBLOCK 接口	144
2.9. 电源供应	150
2.9.1. 数字IO供电 (IOVDD)	151

2.9.2. Digital Core Supply (DVDD)	151
2.9.3. On-Chip Voltage Regulator Input Supply (VREG_VIN)	151
2.9.4. USB PHY Supply (USB_VDD)	151
2.9.5. ADC Supply (ADC_AVDD)	152
2.9.6. Power Supply Sequencing	152
2.9.7. Power Supply Schemes	152
2.10. Core Supply Regulator	155
2.10.1. Application Circuit	155
2.10.2. Operating Modes	156
2.10.3. Output Voltage Select	157
2.10.4. Status	157
2.10.5. Current Limit	157
2.10.6. List of Registers	157
2.10.7. Detailed Specifications	160
2.11. Power Control	160
2.11.1. Top-level Clock Gates	160
2.11.2. SLEEP State	161
2.11.3. DORMANT State	161
2.11.4. Memory Power Down	161
2.11.5. Programmer's Model	162
2.12. Chip-Level Reset	163
2.12.1. Overview	163
2.12.2. Power-on Reset	164
2.12.3. Brown-out Detection	165
2.12.4. Supply Monitor	167
2.12.5. External Reset	167
2.12.6. Rescue Debug Port Reset	167
2.12.7. Source of Last Reset	168
2.12.8. List of Registers	168
2.13. Power-On State Machine	168
2.13.1. Overview	168
2.13.2. Power On Sequence	168
2.13.3. Register Control	169
2.13.4. Interaction with Watchdog	169
2.13.5. List of Registers	169
2.14. Subsystem Resets	172
2.14.1. Overview	172
2.14.2. Programmer's Model	173
2.14.3. List of Registers	175
2.15. Clocks	178
2.15.1. Overview	178
2.15.2. Clock sources	179
2.15.3. Clock Generators	183
2.15.4. Frequency Counter	186
2.15.5. Resus	187
2.15.6. Programmer's Model	187
2.15.7. List of Registers	194
2.16. Crystal Oscillator (XOSC)	216
2.16.1. Overview	216
2.16.2. Usage	217
2.16.3. Startup Delay	217
2.16.4. XOSC Counter	217
2.16.5. DORMANT mode	218
2.16.6. Programmer's Model	218
2.16.7. List of Registers	219
2.17. Ring Oscillator (ROSC)	221
2.17.1. Overview	221
2.17.2. ROSC/XOSC trade-offs	222
2.17.3. Modifying the frequency	222
2.17.4. ROSC divider	223

2.9.2. 数字核心供电 (DVDD)	151
2.9.3. 片上电压调节器输入电源 (VREG_VIN)	151
2.9.4. USB PHY 供电 (USB_VDD)	151
2.9.5. ADC 供电 (ADC_AVDD)	152
2.9.6. 电源顺序	152
2.9.7. 电源方案	152
2.10. 核心供电调节器	155
2.10.1. 应用电路	155
2.10.2. 工作模式	156
2.10.3. 输出电压选择	157
2.10.4. 状态	157
2.10.5. 电流限制	157
2.10.6. 寄存器列表	157
2.10.7. 详细规格	160
2.11. 电源控制	160
2.11.1. 顶层时钟门控	160
2.11.2. SLEEP 状态	161
2.11.3. 休眠状态	161
2.11.4. 内存断电	161
2.11.5. 程序员模型	162
2.12. 芯片级复位	163
2.12.1. 概述	163
2.12.2. 上电复位	164
2.12.3. 欠压检测	165
2.12.4. 电源监控	167
2.12.5. 外部复位	167
2.12.6. 救援调试端口复位	167
2.12.7. 最近一次复位来源	168
2.12.8. 寄存器列表	168
2.13. 上电状态机	168
2.13.1. 概述	168
2.13.2. 上电序列	168
2.13.3. 寄存器控制	169
2.13.4. 与看门狗的交互	169
2.13.5. 寄存器列表	169
2.14. 子系统复位	172
2.14.1. 概述	172
2.14.2. 程序员模型	173
2.14.3. 寄存器列表	175
2.15. 时钟	178
2.15.1. 概述	178
2.15.2. 时钟源	179
2.15.3. 时钟发生器	183
2.15.4. 频率计数器	186
2.15.5. Resus	187
2.15.6. 程序员模型	187
2.15.7. 寄存器列表	194
2.16. 晶体振荡器 (XOSC)	216
2.16.1. 概述	216
2.16.2. 使用方法	217
2.16.3. 启动延迟	217
2.16.4. XOSC 计数器	217
2.16.5. 休眠模式	218
2.16.6. 程序员模型	218
2.16.7. 寄存器列表	219
2.17. 环形振荡器 (ROSC)	221
2.17.1. 概述	221
2.17.2. ROSC/XOSC 折衷	222
2.17.3. 频率调整	222
2.17.4. ROSC 分频器	223

2.17.5. Random Number Generator	223
2.17.6. ROSC Counter	223
2.17.7. DORMANT mode	223
2.17.8. List of Registers	224
2.18. PLL	228
2.18.1. Overview	228
2.18.2. Calculating PLL parameters	228
2.18.3. Configuration	232
2.18.4. List of Registers	234
2.19. GPIO	236
2.19.1. Overview	236
2.19.2. Function Select	237
2.19.3. Interrupts	239
2.19.4. Pads	240
2.19.5. Software Examples	241
2.19.6. List of Registers	244
2.20. Sysinfo	305
2.20.1. Overview	305
2.20.2. List of Registers	305
2.21. Syscfg	306
2.21.1. Overview	306
2.21.2. List of Registers	306
2.22. TBMAN	309
2.22.1. List of Registers	309
3. PIO	311
3.1. Overview	311
3.2. Programmer's Model	312
3.2.1. PIO Programs	312
3.2.2. Control Flow	313
3.2.3. Registers	314
3.2.4. Stalling	317
3.2.5. Pin Mapping	318
3.2.6. IRQ Flags	318
3.2.7. Interactions Between State Machines	318
3.3. PIO Assembler (pioasm)	319
3.3.1. Directives	319
3.3.2. Values	320
3.3.3. Expressions	320
3.3.4. Comments	320
3.3.5. Labels	320
3.3.6. Instructions	321
3.3.7. Pseudoinstructions	321
3.4. Instruction Set	321
3.4.1. Summary	321
3.4.2. JMP	322
3.4.3. WAIT	323
3.4.4. IN	324
3.4.5. OUT	325
3.4.6. PUSH	326
3.4.7. PULL	327
3.4.8. MOV	328
3.4.9. IRQ	329
3.4.10. SET	330
3.5. Functional Details	331
3.5.1. Side-set	331
3.5.2. Program Wrapping	332
3.5.3. FIFO Joining	334
3.5.4. Autopush and Autopull	335
3.5.5. Clock Dividers	339
3.5.6. GPIO Mapping	340

2.17.5. 随机数生成器	223
2.17.6. ROSC 计数器	223
2.17.7. 休眠模式	223
2.17.8. 寄存器列表	224
2.18. PLL	228
2.18.1. 概述	228
2.18.2. PLL 参数计算	228
2.18.3. 配置	232
2.18.4. 寄存器列表	234
2.19. GPIO	236
2.19.1. 概述	236
2.19.2. 功能选择	237
2.19.3. 中断	239
2.19.4. 引脚	240
2.19.5. 软件示例	241
2.19.6. 寄存器列表	244
2.20. 系统信息	305
2.20.1. 概述	305
2.20.2. 寄存器列表	305
2.21. Syscfg	306
2.21.1. 概述	306
2.21.2. 寄存器列表	306
2.22. TBMAN	309
2.22.1. 寄存器列表	309
3. PIO	311
3.1. 概述	311
3.2. 程序员模型	312
3.2.1. PIO 程序	312
3.2.2. 控制流	313
3.2.3. 寄存器	314
3.2.4. 阻塞	317
3.2.5. 引脚映射	318
3.2.6. IRQ 标志	318
3.2.7. 状态机之间的交互	318
3.3. PIO 汇编器 (pioasm)	319
3.3.1. 指令	319
3.3.2. 值	320
3.3.3. 表达式	320
3.3.4. 注释	320
3.3.5. 标签	320
3.3.6. 指令	321
3.3.7. 伪指令	321
3.4. 指令集	321
3.4.1. 概要	321
3.4.2. JMP	322
3.4.3. WAIT	323
3.4.4. IN	324
3.4.5. OUT	325
3.4.6. PUSH	326
3.4.7. PULL	327
3.4.8. MOV	328
3.4.9. IRQ	329
3.4.10. SET	330
3.5. 功能细节	331
3.5.1. 旁路设定	331
3.5.2. 程序封装	332
3.5.3. FIFO 连接	334
3.5.4. 自动推送与自动拉取	335
3.5.5. 时钟分频器	339
3.5.6. GPIO 映射	340

3.5.7. Forced and EXEC'd Instructions.....	342
3.6. Examples	344
3.6.1. Duplex SPI	344
3.6.2. WS2812 LEDs.....	348
3.6.3. UART TX	350
3.6.4. UART RX	352
3.6.5. Manchester Serial TX and RX.....	355
3.6.6. Differential Manchester (BMC) TX and RX.....	357
3.6.7. I2C	361
3.6.8. PWM	364
3.6.9. Addition	366
3.6.10. Further Examples	367
3.7. List of Registers	368
4. Peripherals	383
4.1. USB	383
4.1.1. Overview	383
4.1.2. Architecture	384
4.1.3. Programmer's Model.....	394
4.1.4. List of Registers	398
References	417
4.2. UART	417
4.2.1. Overview	417
4.2.2. Functional description.....	418
4.2.3. Operation	420
4.2.4. UART hardware flow control	422
4.2.5. UART DMA Interface	424
4.2.6. Interrupts	425
4.2.7. Programmer's Model.....	427
4.2.8. List of Registers	429
4.3. I2C	440
4.3.1. Features	440
4.3.2. IP Configuration	441
4.3.3. I2C Overview	441
4.3.4. I2C Terminology.....	443
4.3.5. I2C Behaviour	444
4.3.6. I2C Protocols	445
4.3.7. Tx FIFO Management and START, STOP and RESTART Generation.....	448
4.3.8. Multiple Master Arbitration	450
4.3.9. Clock Synchronization	451
4.3.10. Operation Modes	452
4.3.11. Spike Suppression	457
4.3.12. Fast Mode Plus Operation	458
4.3.13. Bus Clear Feature	458
4.3.14. IC_CLK Frequency Configuration	459
4.3.15. DMA Controller Interface	463
4.3.16. Operation of Interrupt Registers	464
4.3.17. List of Registers	464
4.4. SPI	501
4.4.1. Overview	502
4.4.2. Functional Description	502
4.4.3. Operation	505
4.4.4. List of Registers	515
4.5. PWM	521
4.5.1. Overview	521
4.5.2. Programmer's Model	522
4.5.3. List of Registers	529
4.6. Timer	534
4.6.1. Overview	534
4.6.2. Counter	535
4.6.3. Alarms	535

3.5.7. 强制执行及 EXEC 指令	342
3.6. 示例	344
3.6.1. 双工 SPI	344
3.6.2. WS2812 LED	348
3.6.3. UART 发送	350
3.6.4. UART 接收	352
3.6.5. 曼彻斯特编码串行发送与接收	355
3.6.6. 差分曼彻斯特编码 (BMC) 发送与接收	357
3.6.7. I2C	361
3.6.8. PWM	364
3.6.9. 加法	366
3.6.10. 更多示例	367
3.7. 寄存器列表	368
4. 外设	383
4.1. USB	383
4.1.1. 概述	383
4.1.2. 架构	384
4.1.3. 程序员模型	394
4.1.4. 寄存器列表	398
参考文献	417
4.2. UART	417
4.2.1. 概述	417
4.2.2. 功能描述	418
4.2.3. 操作	420
4.2.4. UART 硬件流控制	422
4.2.5. UART DMA 接口	424
4.2.6. 中断	425
4.2.7. 程序员模型	427
4.2.8. 寄存器列表	429
4.3. I2C	440
4.3.1. 特性	440
4.3.2. IP 配置	441
4.3.3. I2C 概述	441
4.3.4. I2C 术语	443
4.3.5. I2C 行为	444
4.3.6. I2C 协议	445
4.3.7. 发送 FIFO 管理及 START、STOP 和 RESTART 生成	448
4.3.8. 多主机仲裁	450
4.3.9. 时钟同步	451
4.3.10. 操作模式	452
4.3.11. 尖峰抑制	457
4.3.12. 快速模式 Plus 操作	458
4.3.13. 总线清除功能	458
4.3.14. IC_CLK 频率配置	459
4.3.15. DMA 控制器接口	463
4.3.16. 中断寄存器操作	464
4.3.17. 寄存器列表	464
4.4. SPI	501
4.4.1. 概述	502
4.4.2. 功能描述	502
4.4.3. 操作	505
4.4.4. 寄存器列表	515
4.5. PWM	521
4.5.1. 概述	521
4.5.2. 程序员模型	522
4.5.3. 寄存器列表	529
4.6. 定时器	534
4.6.1. 概述	534
4.6.2. 计数器	535
4.6.3. 报警	535

4.6.4. Programmer's Model.....	536
4.6.5. List of Registers.....	539
4.7. Watchdog.....	544
4.7.1. Overview.....	544
4.7.2. Tick generation.....	544
4.7.3. Watchdog Counter.....	545
4.7.4. Scratch Registers.....	545
4.7.5. Programmer's Model.....	545
4.7.6. List of Registers.....	547
4.8. RTC.....	548
4.8.1. Storage Format.....	548
4.8.2. Leap year.....	549
4.8.3. Interrupts.....	549
4.8.4. Reference clock.....	549
4.8.5. Programmer's Model.....	550
4.8.6. List of Registers.....	553
4.9. ADC and Temperature Sensor.....	557
4.9.1. ADC controller.....	558
4.9.2. SAR ADC.....	559
4.9.3. ADC ENOB.....	561
4.9.4. INL and DNL.....	562
4.9.5. Temperature Sensor.....	563
4.9.6. List of Registers.....	564
4.10. SSI.....	567
4.10.1. Overview.....	568
4.10.2. Features.....	568
4.10.3. IP Modifications.....	569
4.10.4. Clock Ratios.....	570
4.10.5. Transmit and Receive FIFO Buffers.....	571
4.10.6. 32-Bit Frame Size Support.....	572
4.10.7. SSI Interrupts.....	572
4.10.8. Transfer Modes.....	573
4.10.9. Operation Modes.....	574
4.10.10. Partner Connection Interfaces.....	579
4.10.11. DMA Controller Interface.....	595
4.10.12. APB Interface.....	597
4.10.13. List of Registers.....	598
5. Electrical and Mechanical.....	607
5.1. Package.....	607
5.1.1. Thermal characteristics.....	608
5.1.2. Recommended PCB Footprint.....	608
5.1.3. Package markings.....	608
5.2. Storage conditions.....	609
5.3. Solder profile.....	609
5.4. Compliance.....	611
5.5. Pinout.....	611
5.5.1. Pin Locations.....	611
5.5.2. Pin Definitions.....	612
5.5.3. Pin Specifications.....	614
5.6. Power Supplies.....	622
5.7. Power Consumption.....	622
5.7.1. Peripheral power consumption.....	622
5.7.2. Power consumption for typical user cases.....	623
Appendix A: Register Field Types.....	625
Standard types.....	625
RW:.....	625
RO:.....	625
WO:.....	625
Clear types.....	625
SC.....	625

4.6.4. 程序员模型	536
4.6.5. 寄存器列表	539
4.7. 看门狗	544
4.7.1. 概述	544
4.7.2. 时钟节拍生成	544
4.7.3. 看门狗计数器	545
4.7.4. 暂存寄存器	545
4.7.5. 程程序员模型	545
4.7.6. 寄存器列表	547
4.8. 实时时钟 (RTC)	548
4.8.1. 存储格式	548
4.8.2. 闰年	549
4.8.3. 中断	549
4.8.4. 参考时钟	549
4.8.5. 程程序员模型	550
4.8.6. 寄存器列表	553
4.9. 模数转换器 (ADC) 及温度传感器	557
4.9.1. ADC 控制器	558
4.9.2. SAR 型 ADC	559
4.9.3. ADC 有效位数 (ENOB)	561
4.9.4. 积分非线性 (INL) 与差分非线性 (DNL)	562
4.9.5. 温度传感器	563
4.9.6. 寄存器列表	564
4.10. 串行同步接口 (SSI)	567
4.10.1. 概述	568
4.10.2. 特性	568
4.10.3. IP 修改	569
4.10.4. 时钟比率	570
4.10.5. 发送与接收 FIFO 缓冲区	571
4.10.6. 32 位帧尺寸支持	572
4.10.7. SSI 中断	572
4.10.8. 传输模式	573
4.10.9. 操作模式	574
4.10.10. 对端连接接口	579
4.10.11. DMA 控制器接口	595
4.10.12. APB 接口	597
4.10.13. 寄存器列表	598
5. 电气与机械	607
5.1. 封装	607
5.1.1. 热特性	608
5.1.2. 推荐 PCB 布局	608
5.1.3. 封装标记	608
5.2. 存储条件	609
5.3. 焊接曲线	609
5.4. 合规性	611
5.5. 引脚定义	611
5.5.1. 引脚位置	611
5.5.2. 引脚说明	612
5.5.3. 引脚规格	614
5.6. 电源供应	622
5.7. 功耗	622
5.7.1. 外设功耗	622
5.7.2. 典型用户案例功耗	623
附录A：寄存器字段类型	625
标准类型	625
RW:	625
RO:	625
WO:	625
清除类型	625
SC	625

WC	625
FIFO types	626
RWF	626
RF	626
WF	626
Appendix B: Errata	627
Bootrom	627
RP2040-E9	627
RP2040-E14	627
Clocks	628
RP2040-E7	628
RP2040-E10	628
DMA	629
RP2040-E12	629
RP2040-E13	629
GPIO / ADC	630
RP2040-E6	630
RP2040-E11	630
USB	630
RP2040-E2	630
RP2040-E3	631
RP2040-E4	631
RP2040-E5	631
RP2040-E15	633
RP2040-E16	634
Watchdog	634
RP2040-E1	634
XIP Flash	635
RP2040-E8	635
Appendix C: Availability	636
Support	636
Ordering code	636
Documentation Release History	637
20 February 2025	637
15 October 2024	637
02 May 2024	637
02 February 2024	637
14 June 2023	637
03 March 2023	637
01 December 2022	637
30 June 2022	638
17 June 2022	638
04 November 2021	638
03 November 2021	638
30 September 2021	638
23 June 2021	638
07 June 2021	639
13 April 2021	639
07 April 2021	639
05 March 2021	639
23 February 2021	639
01 February 2021	639
26 January 2021	639
21 January 2021	640

WC	625
FIFO 类型	626
RWF	626
RF	626
WF	626
附录 B：勘误	627
Bootrom	627
RP2040-E9	627
RP2040-E14	627
时钟	628
RP2040-E7	628
RP2040-E10	628
DMA	629
RP2040-E12	629
RP2040-E13	629
GPIO / ADC	630
RP2040-E6	630
RP2040-E11	630
USB	630
RP2040-E2	630
RP2040-E3	631
RP2040-E4	631
RP2040-E5	631
RP2040-E15	633
RP2040-E16	634
看门狗	634
RP2040-E1	634
XIP Flash	635
RP2040-E8	635
附录 C：供应情况	636
支持	636
订购代码	636
文档发行历史	637
2025年2月20日	637
2024年10月15日	637
2024年5月2日	637
2024年2月2日	637
2023年6月14日	637
2023年3月3日	637
2022年12月1日	637
2022年6月30日	638
2022年6月17日	638
2021年11月4日	638
2021年11月3日	638
2021年9月30日	638
2021年6月23日	638
2021年6月7日	639
2021年4月13日	639
2021年4月7日	639
2021年3月5日	639
2021年2月23日	639
2021年2月1日	639
2021年1月26日	639
2021年1月21日	640

Chapter 1. Introduction

Microcontrollers connect the world of software to the world of hardware. They allow developers to write software which interacts with the physical world in the same deterministic, cycle-accurate manner as digital logic. They occupy the bottom left corner of the price/performance space, outselling their more powerful brethren by a factor of ten to one. They are the workhorses that power the digital transformation of our world.

RP2040 is the debut microcontroller from Raspberry Pi. It brings our signature values of high performance, low cost, and ease of use to the microcontroller space.

With a large on-chip memory, symmetric dual-core processor complex, deterministic bus fabric, and rich peripheral set augmented with our unique Programmable I/O (PIO) subsystem, it provides professional users with unrivalled power and flexibility. With detailed documentation, a polished MicroPython port, and a UF2 bootloader in ROM, it has the lowest possible barrier to entry for beginner and hobbyist users.

RP2040 is a stateless device, with support for cached execute-in-place from external QSPI memory. This design decision allows you to choose the appropriate density of non-volatile storage for your application, and to benefit from the low pricing of commodity Flash parts.

RP2040 is manufactured on a modern 40nm process node, delivering high performance, low dynamic power consumption, and low leakage, with a variety of low-power modes to support extended-duration operation on battery power.

Key features:

- Dual ARM Cortex-M0+ @ 133MHz
- 264kB on-chip SRAM in six independent banks
- Support for up to 16MB of off-chip Flash memory via dedicated QSPI bus
- DMA controller
- Fully-connected AHB crossbar
- Interpolator and integer divider peripherals
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, 4 of which can be used as analogue inputs
- Peripherals
 - 2 UARTs
 - 2 SPI controllers
 - 2 I2C controllers
 - 16 PWM channels
 - USB 1.1 controller and PHY, with host and device support
 - 8 PIO state machines

Whatever your microcontroller application, from machine learning to motor control, from agriculture to audio, RP2040 has the performance, feature set, and support to make your product fly.

1.1. Why is the chip called RP2040?

The post-fix numeral on RP2040 comes from the following,

第一章 引言

微控制器连接软件世界与硬件世界。它们使开发者能够以与数字逻辑相同的确定性和周期精度，编写与物理世界交互的软件。它们位于性价比空间的左下角，以十倍于性能更强同类产品的销量领先。

它们是推动我们世界数字化转型的中坚力量。

RP2040是Raspberry Pi首款推出的微控制器。它将我们标志性的高性能、低成本和易用性价值带入微控制器领域。

凭借大容量片上存储器、对称双核处理器架构、确定性总线结构及丰富的外设组合，辅以我们独特的可编程I/O（PIO）子系统，为专业用户提供无与伦比的性能与灵活性。通过详尽的文档、成熟的MicroPython移植版本以及内置于ROM中的UF2引导程序，为初学者和爱好者用户提供了最低的入门门槛。

RP2040 为无状态设备，支持从外部 QSPI 存储器进行缓存执行。该设计决策允许您根据应用需求选择合适容量的非易失性存储器，并享受普通 Flash 器件的低价优势。

RP2040 采用先进的 40nm 工艺制造，提供高性能、低动态功耗和低漏电，具备多种低功耗模式，以支持电池供电下的长时间运行。

主要特性：

- 双核 ARM Cortex-M0+，主频 133MHz
- 六个独立储存区，片上 SRAM 总容量为 264kB
- 通过专用 QSPI 总线，支持最多 16MB 的片外 Flash 存储
- DMA 控制器
- 全连接 AHB 交叉开关
- 插值器与整数除法器外设
- 片上可编程低压差稳压器（LDO），用于生成核心电压
- 两个片上相位锁定环（PLL），用于生成 USB 和核心时钟
- 30 个 GPIO 引脚，其中 4 个可用作模拟输入
- 外设
 - 2 个 UART
 - 2 个 SPI 控制器
 - 2 个 I2C 控制器
 - 16 个 PWM 通道
 - USB 1.1 控制器及 PHY，支持主机和设备
 - 8 个 PIO 状态机

无论您的微控制器应用领域为何，从机器学习到电机控制，从农业到音频，RP2040 均具备足够的性能、功能与支持，助力您的产品腾飞。

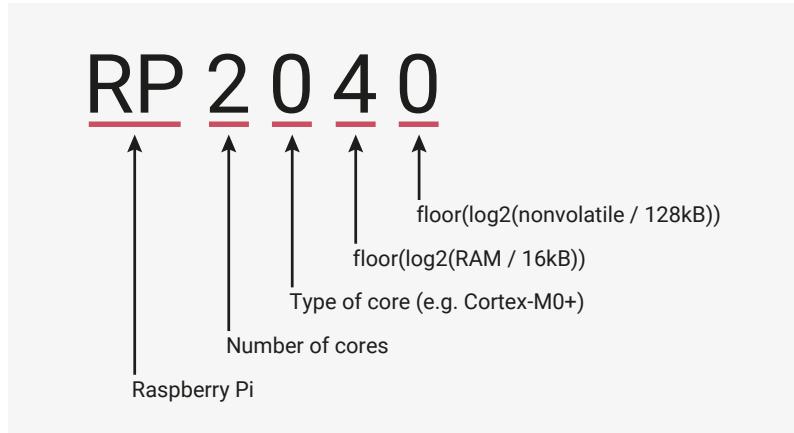
1.1. 芯片为何命名为 RP2040？

RP2040 后缀数字源自以下内容：

1. Number of processor cores (2)
2. Loosely which type of processor (M0+)
3. $\text{floor}(\log_2(\text{RAM} / 16\text{kB}))$
4. $\text{floor}(\log_2(\text{nonvolatile} / 128\text{kB}))$ or 0 if no onboard nonvolatile storage

see [Figure 1](#).

Figure 1. An explanation for the name of the RP2040 chip.



1.2. Summary

RP2040 is a low-cost, high-performance microcontroller device with flexible digital interfaces. Key features:

- Dual Cortex M0+ processor cores, up to 133MHz (or 200MHz at 1.15V, see [Section 2.15.3](#))
- 264kB of embedded SRAM in 6 banks
- 30 multifunction GPIO
- 6 dedicated IO for SPI Flash (supporting XIP)
- Dedicated hardware for commonly used peripherals
- Programmable IO for extended peripheral support
- 4 channel ADC with internal temperature sensor, 500ksps, 12-bit conversion
- USB 1.1 Host/Device

1.3. The Chip

RP2040 has a dual M0+ processor cores, DMA, internal memory and peripheral blocks connected via AHB/APB bus fabric.

1. 处理器核心数量 (2)
2. 大致的处理器类型 (M0+)
3. $\lfloor \log_2(\text{内存} / 16\text{kB}) \rfloor$
4. $\lfloor \log_2(\text{非易失性存储} / 128\text{kB}) \rfloor$, 无板载非易失性存储时为 0

参见图 1。

图1。RP2040芯片名称说明。



1.2. 概要

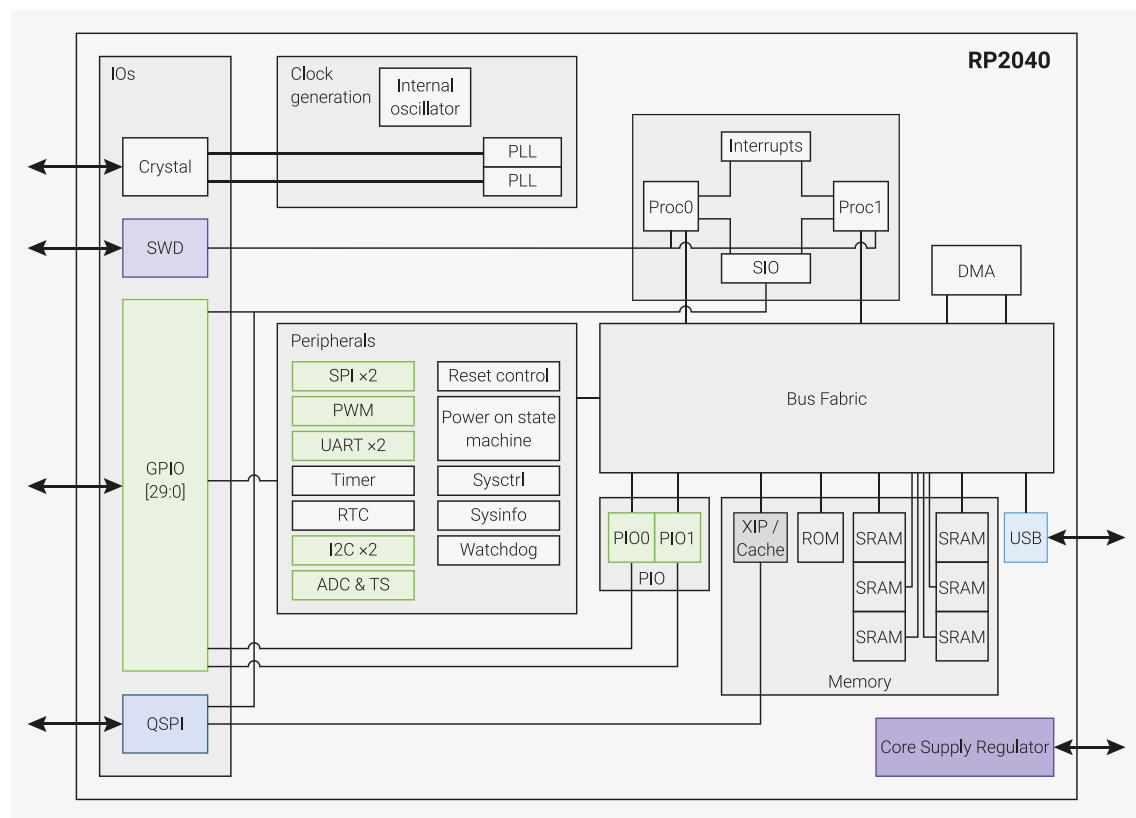
RP2040是一款低成本、高性能的微控制器设备，具备灵活的数字接口。主要特性包括：

- 双核Cortex M0+处理器，最高133MHz（或在1.15V时达200MHz，详见第2.15.3节）
- 264kB嵌入式SRAM，分为6个存储块
- 30个多功能GPIO口
- 6个专用SPI Flash IO端口（支持XIP）
- 为常用外设设计的专用硬件模块
- 可编程IO，支持扩展外设功能
- 4通道ADC，内置温度传感器，500ksps，12位转换精度
- USB 1.1 主机/设备模式

1.3. 芯片

RP2040集成了双核M0+处理器、DMA、内部存储器及通过AHB/APB总线连接的外设模块。

Figure 2. A system overview of the RP2040 chip



Code may be executed directly from external memory through a dedicated SPI, DSPI or QSPI interface. A small cache improves performance for typical applications.

Debug is available via the SWD interface.

Internal SRAM can contain code or data. It is addressed as a single 264 kB region, but physically partitioned into 6 banks to allow simultaneous parallel access from different masters.

DMA bus masters are available to offload repetitive data transfer tasks from the processors.

GPIO pins can be driven directly, or from a variety of dedicated logic functions.

Dedicated hardware for fixed functions such as SPI, I2C, UART.

Flexible configurable PIO controllers can be used to provide a wide variety of IO functions.

A USB controller with embedded PHY can be used to provide FS/LS Host or Device connectivity under software control.

Four ADC inputs which are shared with GPIO pins.

Two PLLs to provide a fixed 48MHz clock for USB or ADC, and a flexible system clock up to 133MHz.

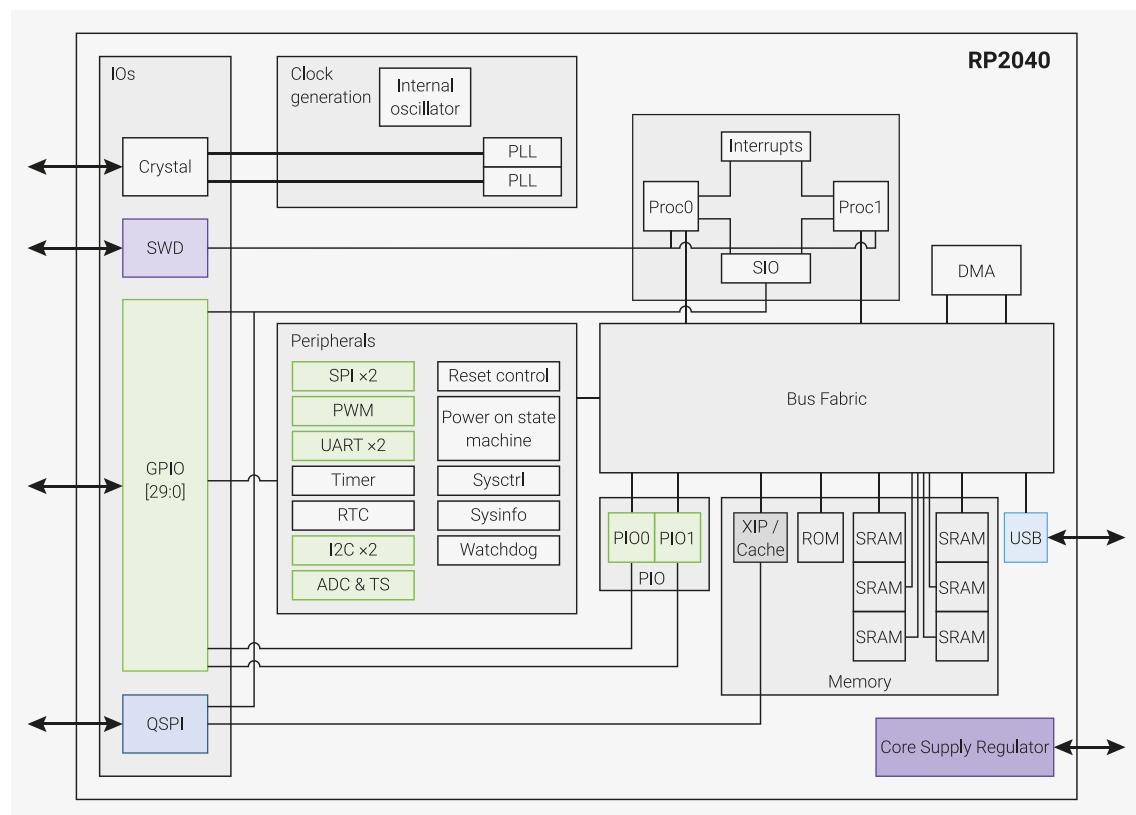
An internal Voltage Regulator to supply the core voltage so the end product only needs supply the IO voltage.

1.4. Pinout Reference

This section provides a quick reference for pinout and pin functions. Full details, including electrical specifications and package drawings, can be found in [Chapter 5](#).

1.4.1. Pin Locations

图2。 RP2040芯
片系统概述



代码可通过专用的 SPI、DSPI 或 QSPI 接口直接从外部存储器执行。小缓存可提升典型应用的性能。

可通过 SWD 接口实现调试功能。

内部 SRAM 可用于存储代码或数据。其地址空间为单一 264 kB 区域，但物理上划分为 6 个存储块，以允许不同主控单元的并行访问。

DMA 总线主控可用以分担处理器的重复数据传输任务。

GPIO 引脚可直接驱动，或由多种专用逻辑功能驱动。

设有专用硬件支持的固定功能，如 SPI、I2C、UART。

灵活可配置的 PIO 控制器可用于实现多种 IO 功能。

嵌入式 PHY 的 USB 控制器可在软件控制下提供 FS/LS 主机或设备连接。

四个 ADC 输入端口，与 GPIO 引脚共享。

两个 PLL，用于提供 USB 或 ADC 的固定 48MHz 时钟，以及最高 133MHz 的灵活系统时钟。

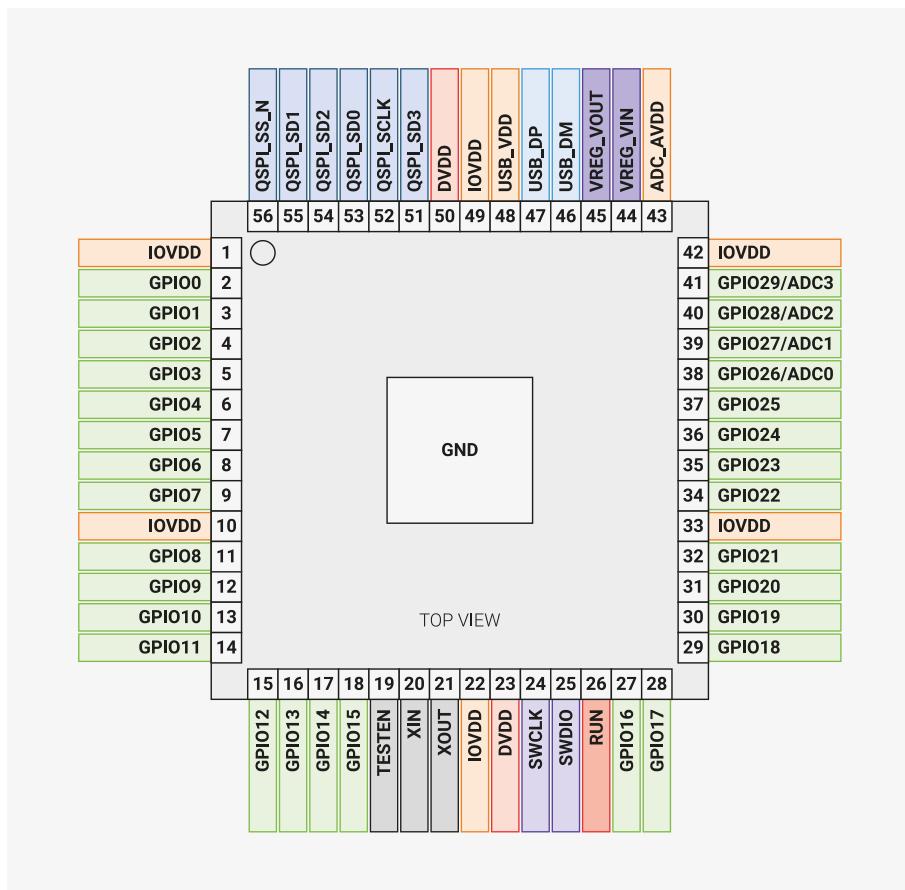
内部电压调节器负责提供核心电压，因此最终产品仅需提供 IO 电压。

1.4. 引脚参考

本节提供引脚排列及引脚功能的快速参考。完整详情，包括电气规格及封装图，详见第5章。

1.4.1. 引脚位置

Figure 3. RP2040
Pinout for QFN-56
7x7mm (reduced ePad
size)

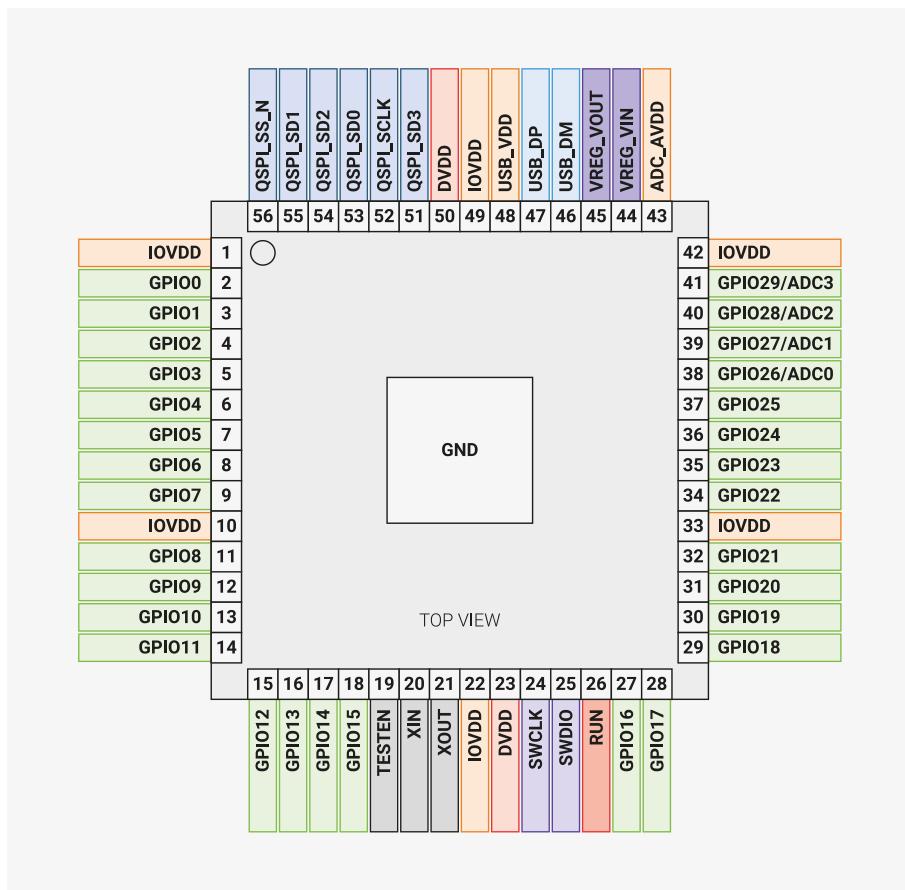


1.4.2. Pin Descriptions

Table 1. The function of each pin is briefly described here. Full electrical specifications can be found in Chapter 5.

Name	Description
GPIOx	General-purpose digital input and output. RP2040 can connect one of a number of internal peripherals to each GPIO, or control GPIOs directly from software.
GPIOx/ADCy	General-purpose digital input and output, with analogue-to-digital converter function. The RP2040 ADC has an analogue multiplexer which can select any one of these pins, and sample the voltage.
QSPIx	Interface to a SPI, Dual-SPI or Quad-SPI flash device, with execute-in-place support. These pins can also be used as software-controlled GPIOs, if they are not required for flash access.
USB_DM and USB_DP	USB controller, supporting Full Speed device and Full/Low Speed host. A 27Ω series termination resistor is required on each pin, but bus pull-ups and pull-downs are provided internally.
XIN and XOUT	Connect a crystal to RP2040's crystal oscillator. XIN can also be used as a single-ended CMOS clock input, with XOUT disconnected. The USB bootloader requires a 12MHz crystal or 12MHz clock input. For recommended crystals, see Crystal Oscillator (Section 2.16).
RUN	Global asynchronous reset pin. Reset when driven low, run when driven high. If no external reset is required, this pin can be tied directly to IOVDD.
SWCLK and SWDIO	Access to the internal Serial Wire Debug multi-drop bus. Provides debug access to both processors, and can be used to download code.
TESTEN	Factory test mode pin. Tie to GND.
GND	Single external ground connection, bonded to a number of internal ground pads on the RP2040 die.
IOVDD	Power supply for digital GPIOs, nominal voltage 1.8V to 3.3V

图3。QFN-56 7x7mm
(减小ePad尺寸)
的RP2040引脚
排列



1.4.2. 引脚描述

表1。此处简要描述各引脚功能。
完整电气规格详见第5章。
。

名称	描述
GPIOx	通用数字输入与输出。RP2040可将多个内部外设之一连接至每个GPIO，或通过软件直接控制GPIO。
GPIOx/ADCy	通用数字输入与输出，具备模数转换功能。RP2040的ADC配备模拟多路复用器，可选择任一引脚进行电压采样。
QSPIx	接口用于SPI、双SPI或四路SPI闪存设备，支持执行在位功能。若不用于闪存访问，此类引脚亦可作为软件控制的GPIO使用。
USB_DM 和 USB_DP	USB控制器，支持全速设备及全速/低速主机。每个引脚须配备27Ω串联终端电阻，但总线的上拉和下拉电阻由内部提供。
XIN 和 XOUT	将晶体连接至RP2040的晶体振荡器。XIN亦可用作单端CMOS时钟输入，需断开XOUT。USB引导加载程序要求12MHz晶体或12MHz时钟输入。推荐晶体详见晶体振荡器（第2.16节）。
RUN	全局异步复位引脚。置低复位，置高运行。若无需外部复位，该引脚可直接连接至IOVDD。
SWCLK 和 SWDIO	访问内部串行线调试多路总线。提供对两处理器的调试访问，亦可用于代码下载。
测试	工厂测试模式引脚，连接至地线(GND)。
GND	单一外部地线连接，连接至RP2040芯片内部多个地线焊盘。
IOVDD	数字GPIO电源，额定电压1.8V至3.3V。

Name	Description
USB_VDD	Power supply for internal USB Full Speed PHY, nominal voltage 3.3V
ADC_AVDD	Power supply for analogue-to-digital converter, nominal voltage 3.3V
VREG_VIN	Power input for the internal core voltage regulator, nominal voltage 1.8V to 3.3V
VREG_VOUT	Power output for the internal core voltage regulator, nominal voltage 1.1V, 100mA max current
DVDD	Digital core power supply, nominal voltage 1.1V. Can be connected to VREG_VOUT, or to some other board-level power supply.

1.4.3. GPIO Functions

Each individual GPIO pin can be connected to an internal peripheral via the GPIO functions defined below. Some internal peripheral connections appear in multiple places to allow some system level flexibility. SIO, PIO0 and PIO1 can connect to all GPIO pins and are controlled by software (or software controlled state machines) so can be used to implement many functions.

Table 2. General Purpose Input/Output (GPIO) Bank 0 Functions

GPIO	Function								
	F1	F2	F3	F4	F5	F6	F7	F8	F9
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB OVCUR DET
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS DET
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB VBUS EN
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB OVCUR DET
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1		USB VBUS DET
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1		USB VBUS EN
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1		USB OVCUR DET
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1		USB VBUS DET
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1		USB VBUS EN
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1		USB OVCUR DET
10	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS DET
11	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB VBUS EN
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB OVCUR DET
13	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS DET
14	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1		USB VBUS EN
15	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1		USB OVCUR DET
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB VBUS DET
17	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS EN
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB OVCUR DET
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB VBUS DET
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0	USB VBUS EN
21	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB OVCUR DET

名称	描述
USB_VDD	内部USB全速PHY电源，额定电压3.3V。
ADC_AVDD	模数转换器电源，额定电压3.3V。
VREG_VIN	内部核心稳压器电源输入，额定电压1.8V至3.3V。
VREG_VOUT	内部核心稳压器电源输出，额定电压1.1V，最大电流100mA。
DVDD	数字核心电源，额定电压1.1V。可连接至VREG_VOUT或其他板级电源。

1.4.3. GPIO 功能

每个GPIO引脚均可通过以下定义的GPIO功能连接至内部外设。部分内部外设连接在多个位置出现，以提供系统级灵活性。SIO、PIO0 和 PIO1 可连接至所有 GPIO 引脚，并由软件（或软件控制状态机）管理，因此可用于实现多种功能。

表2。通用输入/输出 (GPIO) 组功能

GPIO	功能								
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB 过流检测
1	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1		USB 总线电压检测
2	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
3	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1		USB 过流检测
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1		USB 总线电压检测
5	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
6	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1		USB 过流检测
7	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1		USB 总线电压检测
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
9	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1		USB 过流检测
10	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线电压检测
11	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 过流检测
13	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线电压检测
14	SPI1 SCK	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM7 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
15	SPI1 TX	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM7 B	SIO	PIO0	PIO1		USB 过流检测
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB 总线电压检测
17	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
18	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1		USB 过流检测
19	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1		USB 总线电压检测
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0	USB 总线使能 (VBUS EN)
21	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB 过流检测

	Function									
22	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB VBUS DET	
23	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB VBUS EN	
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB OVCUR DET	
25	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB VBUS DET	
26	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS EN	
27	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB OVCUR DET	
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB VBUS DET	
29	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS EN	

Table 3. GPIO bank 0 function descriptions

Function Name	Description
SPIx	Connect one of the internal PL022 SPI peripherals to GPIO
UARTx	Connect one of the internal PL011 UART peripherals to GPIO
I2Cx	Connect one of the internal DW I2C peripherals to GPIO
PWMx A/B	Connect a PWM slice to GPIO. There are eight PWM slices, each with two output channels (A/B). The B pin can also be used as an input, for frequency and duty cycle measurement.
SIO	Software control of GPIO, from the single-cycle IO (SIO) block. The SIO function (F5) must be selected for the processors to <i>drive</i> a GPIO, but the input is always connected, so software can check the state of GPIOs at any time.
PIOx	Connect one of the programmable IO blocks (PIO) to GPIO. PIO can implement a wide variety of interfaces, and has its own internal pin mapping hardware, allowing flexible placement of digital interfaces on bank 0 GPIOs. The PIO function (F6 , F7) must be selected for PIO to <i>drive</i> a GPIO, but the input is always connected, so the PIOs can always see the state of all pins.
CLOCK GPINx	General purpose clock inputs. Can be routed to a number of internal clock domains on RP2040, e.g. to provide a 1Hz clock for the RTC, or can be connected to an internal frequency counter.
CLOCK GPOUTx	General purpose clock outputs. Can drive a number of internal clocks (including PLL outputs) onto GPIOs, with optional integer divide.
USB OVCUR DET/VBUS DET/VBUS EN	USB power control signals to/from the internal USB controller

功能									
22	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB 总线电压检测
23	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB 总线使能 (VBUS EN)
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB 过流检测
25	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB 总线电压检测
26	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
27	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 过流检测
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 总线电压检测
29	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)

表 3。GPIO 组 0
功能描述

功能名称	描述
SPIx	将内部 PL022 SPI 外设之一连接至 GPIO
UARTx	将内部 PL011 UART 外设之一连接至 GPIO
I2Cx	将内部 DW I2C 外设之一连接至 GPIO
PWMx A/B	将PWM切片连接至GPIO。共有八个PWM切片，每个切片包含两个输出通道（A/B）。B引脚亦可用作输入，用于频率和占空比的测量。
SIO	通过单周期IO (SIO) 模块实现GPIO的软件控制。必须选择SIO功能 (F5) 以使处理器驱动GPIO，但输入端始终连接，因此软件能够随时检测GPIO状态。
PIOx	将可编程IO模块 (PIO) 之一连接至GPIO。PIO可实现多种接口，且内置引脚映射硬件，允许数字接口灵活地分配至银行0的GPIO。须选择PIO功能 (F6, F7) 以使PIO驱动GPIO，但输入端始终连接，PIO可持续监测所有引脚状态。
CLOCK GPINx	通用时钟输入。可路由至RP2040的多个内部时钟域，例如为RTC提供1Hz时钟，或连接至内部频率计数器。
CLOCK GPOUTx	通用时钟输出。可将多个内部时钟（包括PLL输出）驱动至GPIO，并支持可选的整数分频。
USB OVCUR DET/VBUS DET/VBUS EN	内部USB控制器的USB电源控制信号。

Chapter 2. System Description

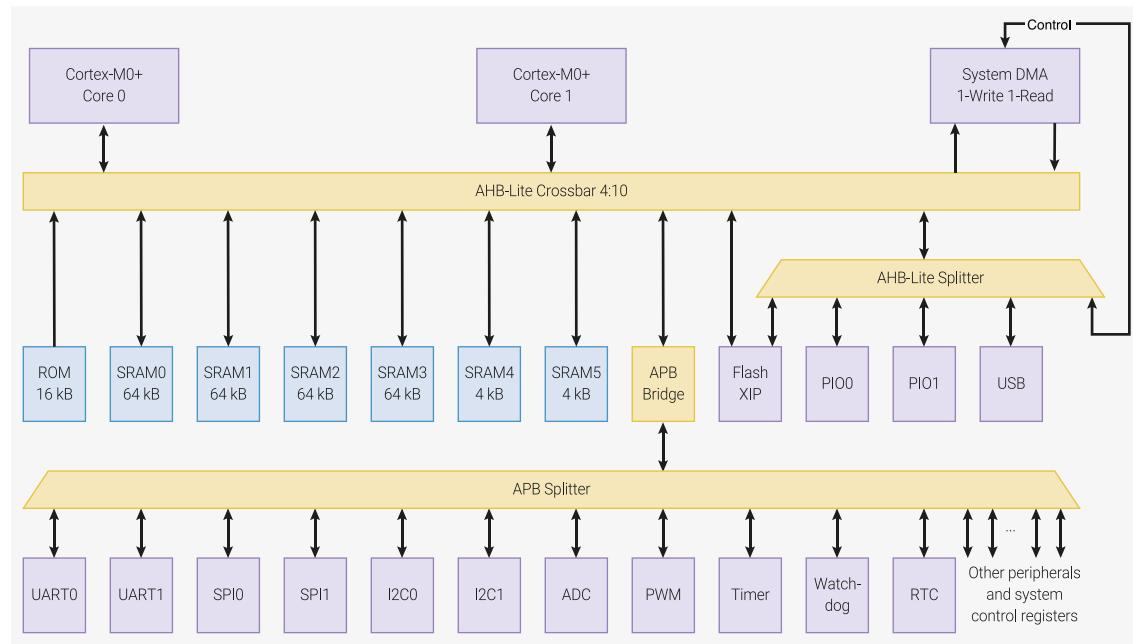
This chapter describes the RP2040 key system features including processor, memory, how blocks are connected, clocks, resets, power, and IO. Refer to [Figure 2](#) for an overview diagram.

2.1. Bus Fabric

The RP2040 bus fabric routes addresses and data across the chip.

[Figure 4](#) shows the high-level structure of the bus fabric. The main AHB-Lite crossbar routes addresses and data between its 4 upstream ports and 10 downstream ports: up to four bus transfers can take place each cycle. All data paths are 32 bits wide. Memory devices have dedicated ports on the main crossbar, to satisfy their high bandwidth requirements. High-bandwidth AHB-Lite peripherals have a shared port on the crossbar, and an APB bridge provides bus access to system control registers and lower-bandwidth peripherals.

Figure 4. RP2040 bus fabric overview.



The bus fabric connects 4 AHB-Lite masters, i.e. devices which generate addresses:

- Processor core 0
- Processor core 1
- DMA controller Read port
- DMA controller Write port

These are routed through to 10 downstream ports on the main crossbar:

- ROM
- Flash XIP
- SRAM 0 to 5 (one port each)
- Fast AHB-Lite peripherals: PIO0, PIO1, USB, DMA control registers, XIP aux (one shared port)
- Bridge to all APB peripherals, and system control registers

The four bus masters can access any four *different* crossbar ports simultaneously, the bus fabric does not add wait

第2章 系统描述

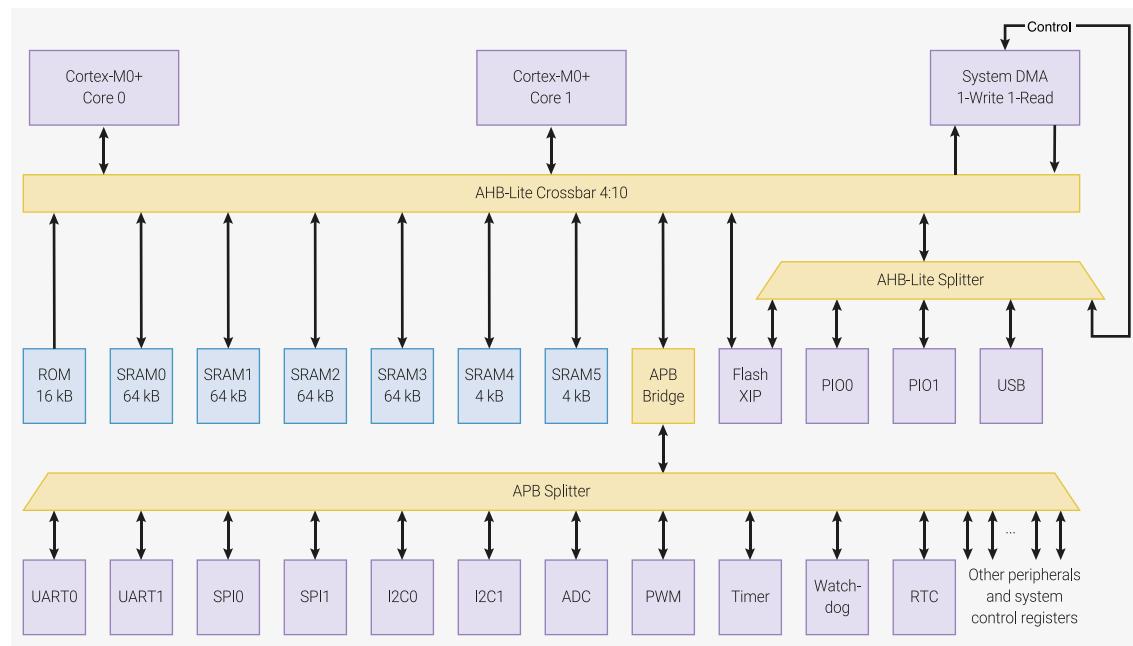
本章介绍RP2040的关键系统特性，包括处理器、内存、模块连接方式、时钟、复位、电源与IO。详见图2的概述图。

2.1. 总线结构

RP2040总线结构负责在芯片内部路由地址和数据信号。

图4展示了总线结构的高级架构。主 AHB-Lite 交叉开关在 4 个上游端口和 10 个下游端口间路由地址与数据：每个时钟周期最多支持四次总线传输。所有数据路径均为 32 位宽。内存设备在主交叉开关上设有专用端口，以满足其高带宽需求。高带宽的 AHB-Lite 外设在交叉开关上共享一个端口，APB 桥提供对系统控制寄存器及低带宽外设的总线访问。

图4。RP2040总线
结构概述



总线结构连接了 4 个 AHB-Lite 主设备，即生成地址的设备：

- 处理器核0
- 处理器核1
- DMA控制器读端口
- DMA控制器写端口

上述设备通过主交叉开关连接至 10 个下游端口：

- ROM
- Flash XIP
- SRAM 0 至 5（各设一个端口）
- 高速 AHB-Lite 外设：PIO0、PIO1、USB、DMA 控制寄存器、XIP 辅助端口（共享一个端口）
- 连接所有 APB 外设及系统控制寄存器的桥接

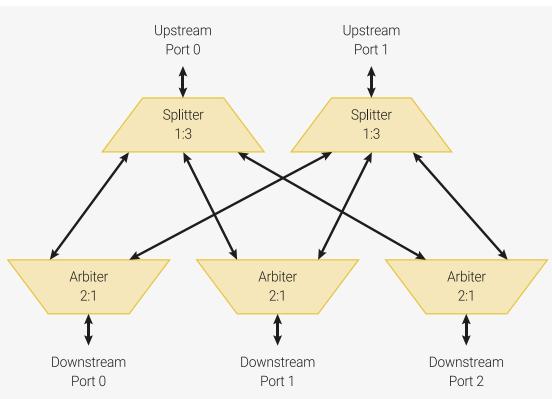
四个总线主设备可同时访问任意四个不同的交叉开关端口，总线结构不引入等待

states to any AHB-Lite slave access. So at a system clock of 125MHz the maximum sustained bus bandwidth is 2.0GBps. The system address map has been arranged to make this parallel bandwidth available to as many software use cases as possible – for example, the striped SRAM alias ([Section 2.6.2](#)) scatters main memory accesses across four crossbar ports (SRAM0...3), so that more memory accesses can proceed in parallel.

2.1.1. AHB-Lite Crossbar

At the centre of the RP2040 bus fabric is a 4:10 fully-connected crossbar. Its 4 upstream ports are connected to the 4 system bus masters, and the 10 downstream ports connect to the highest-bandwidth AHB-Lite slaves (namely the memory interfaces) and to lower layers of the fabric. [Figure 5](#) shows the structure of a 2:3 AHB-Lite crossbar, arranged identically to the 4:10 crossbar on RP2040, but easier to show in the diagram.

Figure 5. A 2:3 AHB-Lite crossbar. Each upstream port connects to a splitter, which routes bus requests toward one of the 3 downstream ports, and routes responses back. Each downstream port connects to an arbiter, which safely manages concurrent access to the port.



The crossbar is built from two components:

- Splitters
 - Perform coarse address decode
 - Route requests (addresses, write data) to the downstream port indicated by the initial address decode
 - Route responses (read data, bus errors) from the correct arbiter back to the upstream port
- Arbiters
 - Manage concurrent requests to a downstream port
 - Route responses (read data, bus errors) to the correct splitter
 - Implement bus priority rules

The main crossbar on RP2040 consists of 4 1:10 splitters and 10 4:1 arbiters, with a mesh of 40 AHB-Lite bus channels between them. Note that, as AHB-Lite is a pipelined bus, the splitter may be routing back a response to an earlier request from downstream port A, whilst a new request to downstream port B is already in progress. This does not incur any cycle penalty.

2.1.1.1. Bus Priority

The arbiters in the main AHB-Lite crossbar implement a two-level bus priority scheme. Priority levels are configured per-master, using the `BUS_PRIORITY` register in the `BUSCTRL` register block.

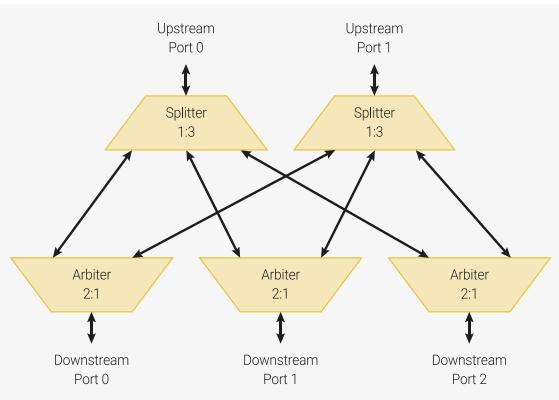
When there are multiple simultaneous accesses to same arbiter, any requests from high-priority masters (priority level 1) will be considered before any requests from low-priority masters (priority 0). If multiple masters of the same priority level attempt to access the same slave simultaneously, a round-robin tie break is applied, i.e. the arbiter grants access to each master in turn.

针对任何AHB-Lite从设备访问的状态。因此，在系统时钟为125MHz时，最大持续总线带宽为2.0GBps。系统地址映射已被设计为使该并行带宽可服务于尽可能多的软件应用场景 —— 例如，条带化SRAM别名（第2.6.2节）将主存访问分散到四个交叉开关端口（[SRAM0...3](#)），以使更多存储访问能够并行进行。

2.1.1. AHB-Lite 交叉开关

RP2040总线结构的核心是一个4:10全连接交叉开关。其4个上游端口连接至4个系统总线主设备，10个下游端口连接至带宽最高的AHB-Lite从设备（即内存接口）及总线结构的下层部分。图5展示了2:3的AHB-Lite交叉开关结构，排列与RP2040上的4:10交叉开关相同，但图中展示更为简洁。

图5。一个2:3的AHB-Lite交叉开关。
每个上游端口连接至一个分配器，该分配器将总线请求引导至三个下游端口中的一个，并回传响应。每个下游端口连接至一个仲裁器，安全管理对该端口的并发访问。



交叉开关由以下两个组件构成：

- 分配器
 - 执行粗地址解码
 - 将请求（地址、写入数据）引导至初始地址解码指示的下游端口
 - 将来自正确仲裁器的响应（读取数据、总线错误）回传至上游端口
- 仲裁器
 - 管理对下游端口的并发请求
 - 将响应（读取数据、总线错误）引导至正确的分配器
 - 实施总线优先级规则

RP2040上的主交叉开关由4个1:10分配器和10个4:1仲裁器组成，彼此之间由40条AHB-Lite总线通道连接。注意，由于AHB-Lite为流水线总线，分配器可能在回传针对下游端口A的先前请求响应时，同时处理对下游端口B的新请求。这不会引起任何周期惩罚。

2.1.1.1. 总线优先级

主AHB-Lite交叉开关中的仲裁器实现了两级总线优先级方案。优先级等级按主控单元配置，使用BUSCTRL寄存器组中的US_PRIORITY寄存器。

当多个访问请求同时到达同一仲裁器时，来自高优先级主控单元（优先级等级为1）的请求将优先于低优先级主控单元（优先级等级为0）的请求。若多个相同优先级的主控单元同时试图访问同一从设备，将采用轮询方式进行平局处理，即仲裁器依次赋予各主控单元访问权限。

NOTE

Priority arbitration only applies to multiple masters attempting to access the **same** slave on the same cycle. Accesses to different slaves, e.g. different SRAM banks, can proceed simultaneously.

When accessing a slave with zero wait states, such as SRAM (i.e. can be accessed once per system clock cycle), high-priority masters will never observe any slowdown or other timing effects caused by accesses from low-priority masters. This allows *guaranteed* latency and throughput for hard real time use cases; it does however mean a low-priority master may get stalled until there is a free cycle.

2.1.1.2. Bus Performance Counters

The performance counters automatically count accesses to the main AHB-Lite crossbar arbiters. This can assist in diagnosing performance issues, in high-traffic use cases.

There are four performance counters. Each is a 24-bit saturating counter. Counter values can be read from `BUSCTRL_PERFCTRx`, and cleared by writing any value to `BUSCTRL_PERFCTRx`. Each counter can count one of the 20 available events at a time, as selected by `BUSCTRL_PERFSELx`. The available bus events are:

PERFSEL _x	Event	Description
0	APB access, contested	Completion of an access to the APB arbiter (which is upstream of all APB peripherals), which was previously delayed due to an access by another master.
1	APB access	Completion of an access to the APB arbiter
2	FASTPERI access, contested	Completion of an access to the FASTPERI arbiter (which is upstream of PIOs, DMA config port, USB, XIP aux FIFO port), which was previously delayed due to an access by another master.
3	FASTPERI access	Completion of an access to the FASTPERI arbiter
4	SRAM5 access, contested	Completion of an access to the SRAM5 arbiter, which was previously delayed due to an access by another master.
5	SRAM5 access	Completion of an access to the SRAM5 arbiter
6	SRAM4 access, contested	Completion of an access to the SRAM4 arbiter, which was previously delayed due to an access by another master.
7	SRAM4 access	Completion of an access to the SRAM4 arbiter
8	SRAM3 access, contested	Completion of an access to the SRAM3 arbiter, which was previously delayed due to an access by another master.
9	SRAM3 access	Completion of an access to the SRAM3 arbiter
10	SRAM2 access, contested	Completion of an access to the SRAM2 arbiter, which was previously delayed due to an access by another master.
11	SRAM2 access	Completion of an access to the SRAM2 arbiter
12	SRAM1 access, contested	Completion of an access to the SRAM1 arbiter, which was previously delayed due to an access by another master.
13	SRAM1 access	Completion of an access to the SRAM1 arbiter
14	SRAM0 access, contested	Completion of an access to the SRAM0 arbiter, which was previously delayed due to an access by another master.
15	SRAM0 access	Completion of an access to the SRAM0 arbiter

注意

优先级仲裁仅适用于多个主控单元在同一周期尝试访问同一从设备的情况。
访问不同的从设备，例如不同的SRAM存储区，可以同时进行。

当访问零等待状态的从设备时，如SRAM（即每个系统时钟周期可访问一次），高优先级主控单元不会因低优先级主控单元的访问而受到任何延迟或其他时序影响。
这允许保证硬实时使用场景下的延迟和吞吐量；但这意味着低优先级主设备可能会被阻塞，直到出现空闲周期。

2.1.1.2. 总线性能计数器

性能计数器自动统计对主AHB-Lite交叉开关仲裁器的访问次数。这有助于诊断高流量使用场景中的性能问题。

共有四个性能计数器。每个计数器为24位饱和计数器。计数器数值可通过`BUSCTRL_PERFCTRx`读取，向`BUSCTRL_PERFCTRx`写入任意值可清零。每个计数器一次可计数20个可选事件中的一个，由`BUSCTRL_PERFSELx`配置。可用总线事件如下：

PERFSEL _x	事件	描述
0	APB访问，存在竞争	完成对APB仲裁器（位于所有APB外设上游）的访问，该访问曾因另一主设备的访问而延迟。
1	APB访问	对APB仲裁器访问的完成
2	FASTPERI访问，存在竞争	对FASTPERI仲裁器的访问完成（此仲裁器位于PIO、DMA配置端口、USB及XIP辅助FIFO端口的上游），此前因其他主控访问而被延迟。
3	FASTPERI访问	对FASTPERI仲裁器访问的完成
4	SRAM5访问，存在竞争	对SRAM5仲裁器的访问完成，此前因其他主控访问而被延迟。
5	SRAM5访问	对SRAM5仲裁器访问的完成
6	SRAM4访问，存在竞争	对SRAM4仲裁器的访问完成，此前因其他主控访问而被延迟。
7	SRAM4访问	对SRAM4仲裁器访问的完成
8	SRAM3访问，存在竞争	对SRAM3仲裁器的访问完成，此前因其他主控访问而被延迟。
9	SRAM3访问	完成对SRAM3仲裁器的访问
10	SRAM2访问，争用中	完成对SRAM2仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
11	SRAM2访问	完成对SRAM2仲裁器的访问
12	SRAM1访问，争用中	完成对SRAM1仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
13	SRAM1访问	完成对SRAM1仲裁器的访问
14	SRAM0访问，争用中	完成对SRAM0仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
15	SRAM0访问	完成对SRAM0仲裁器的访问

PERFSEL x	Event	Description
16	XIP_MAIN access, contested	Completion of an access to the XIP_MAIN arbiter, which was previously delayed due to an access by another master.
17	XIP_MAIN access	Completion of an access to the XIP_MAIN arbiter
18	ROM access, contested	Completion of an access to the ROM arbiter, which was previously delayed due to an access by another master.
19	ROM access	Completion of an access to the ROM arbiter

2.1.2. Atomic Register Access

Each peripheral register block is allocated 4kB of address space, with registers accessed using one of 4 methods, selected by address decode.

- `Addr + 0x0000` : normal read write access
- `Addr + 0x1000` : atomic XOR on write
- `Addr + 0x2000` : atomic bitmask set on write
- `Addr + 0x3000` : atomic bitmask clear on write

This allows individual fields of a control register to be modified without performing a read-modify-write sequence in software: instead the changes are posted to the peripheral, and performed in-situ. Without this capability, it is difficult to safely access IO registers when an interrupt service routine is concurrent with code running in the foreground, or when the two processors are running code in parallel.

The four atomic access aliases occupy a total of 16kB. Most peripherals on RP2040 provide this functionality natively, and atomic writes have the same timing as normal read/write access. Some peripherals (I2C, UART, SPI and SSI) instead have this functionality added using a bus interposer, which translates upstream atomic writes into downstream read-modify-write sequences, at the boundary of the peripheral. This extends the access time by two system clock cycles.

The SIO ([Section 2.3.1](#)), a single-cycle IO block attached directly to the cores' IO ports, does **not** support atomic accesses at the bus level, although some individual registers (e.g. GPIO) have set/clear/xor aliases.

2.1.3. APB Bridge

The APB bridge interfaces the high-speed main AHB-Lite interconnect to the lower-bandwidth peripherals. Whilst the AHB-Lite fabric offers zero-wait-state access everywhere, APB accesses have a cycle penalty:

- APB bus accesses take two cycles minimum (setup phase and access phase)
- The bridge adds an additional cycle to read accesses, as the bus request and response are registered
- The bridge adds **two** additional cycles to write accesses, as the APB setup phase can not begin until the AHB-Lite write data is valid

As a result, the throughput of the APB portion of the bus fabric is somewhat lower than the AHB-Lite portion. However, there is more than sufficient bandwidth to saturate the APB serial peripherals.

2.1.4. Narrow IO Register Writes

Memory-mapped IO registers on RP2040 ignore the width of bus read/write accesses. They treat all writes as though they were 32 bits in size. This means software can not use byte or halfword writes to modify part of an IO register: any write to an address where the 30 address MSBs match the register address will affect the contents of the entire

PERFSEL x	事件	描述
16	XIP_MAIN 访问， 争用中	完成对 XIP_MAIN 仲裁器的访问，该访问因另一个主控的访问而先前被延迟。
17	XIP_MAIN 访问	完成对 XIP_MAIN 仲裁器的访问
18	ROM访问， 受阻	对ROM仲裁器的访问完成，该访问因另一主控设备的访问而被延迟。
19	ROM访问	对ROM仲裁器的访问完成

2.1.2. 原子寄存器访问

每个外设寄存器块分配了4kB地址空间，寄存器通过以下四种方法之一访问，由地址译码选择。

- 地址 + `0x0000`: 普通读写访问
- 地址 + `0x1000`: 写时原子异或
- 地址 + `0x2000`: 写时原子位掩码置位
- 地址 + `0x3000`: 写时原子位掩码清零

这允许对控制寄存器的单个字段进行修改，无需执行软件中的读-改-写序列；修改直接提交给外设，并现场执行。如果没有此功能，当中断服务例程与前台代码并发运行，或两个处理器并行执行代码时，安全访问IO寄存器将非常困难。

四个原子访问别名共占用16kB空间。RP2040上的大多数外设本身具备此功能，且原子写入的时序与普通读写访问相同。部分外设（I2C、UART、SPI和SSI）通过总线插入设备实现此功能，该设备在外设边界将上游的原子写操作转换为下游的读-改-写序列，从而使访问时间增加两个系统时钟周期。

SIO（第2.3.1节）为直接连接至核IO端口的单周期IO模块，不支持总线级别的原子访问，尽管部分寄存器（例如GPIO）提供设定／清除／异或别名。

2.1.3. APB 桥接

APB桥接器将高速主AHB-Lite互连与低带宽外设连接。虽然AHB-Lite结构在各处均支持零等待态访问，但APB访问存在周期惩罚：

- APB总线访问至少耗时两个周期（设置阶段和访问阶段）。
- 由于总线请求和响应采取寄存方式，桥接器在读访问中增加了一个额外周期。
- 由于APB设置阶段必须等到AHB-Lite写数据有效后才能开始，桥接器在写访问中增加了两个额外周期。

因此，APB部分总线结构的吞吐量略低于AHB-Lite部分。然而，带宽完全足以满足APB串行外设的饱和需求。

2.1.4. 窄带 IO 寄存器写入

RP2040上的内存映射IO寄存器忽略总线读/写访问的宽度。它们将所有写操作视为32位大小。这意味着软件不能使用字节或半字写操作来修改IO寄存器的部分内容：任何地址的30个高位地址与寄存器地址匹配的写操作都会影响整个

register.

To update part of an IO register, without a read-modify-write sequence, the best solution on RP2040 is atomic set/clear/XOR (see [Section 2.1.2](#)). Note that this is more flexible than byte or halfword writes, as any combination of fields can be updated in one operation.

Upon a 8-bit or 16-bit write (such as a `strb` instruction on the Cortex-M0+), an IO register will sample the entire 32-bit write databus. The Cortex-M0+ and DMA on RP2040 will always replicate narrow data across the bus:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/system/narrow_io_write/narrow_io_write.c Lines 19 - 62

```

19 int main() {
20     stdio_init_all();
21
22     // We'll use WATCHDOG_SCRATCH0 as a convenient 32 bit read/write register
23     // that we can assign arbitrary values to
24     io_rw_32 *scratch32 = &watchdog_hw->scratch[0];
25     // Alias the scratch register as two halfwords at offsets +0x0 and +0x2
26     volatile uint16_t *scratch16 = (volatile uint16_t *) scratch32;
27     // Alias the scratch register as four bytes at offsets +0x0, +0x1, +0x2, +0x3:
28     volatile uint8_t *scratch8 = (volatile uint8_t *) scratch32;
29
30     // Show that we can read/write the scratch register as normal:
31     printf("Writing 32 bit value\n");
32     *scratch32 = 0xdeadbeef;
33     printf("Should be 0xdeadbeef: 0x%08x\n", *scratch32);
34
35     // We can do narrow reads just fine -- IO registers treat this as a 32 bit
36     // read, and the processor/DMA will pick out the correct byte lanes based
37     // on transfer size and address LSBs
38     printf("\nReading back 1 byte at a time\n");
39     // Little-endian!
40     printf("Should be ef be ad de: %02x ", scratch8[0]);
41     printf("%02x ", scratch8[1]);
42     printf("%02x ", scratch8[2]);
43     printf("%02x\n", scratch8[3]);
44
45     // Byte writes are replicated four times across the 32-bit bus, and IO
46     // registers usually sample the entire write bus.
47     printf("\nWriting 8 bit value 0xa5 at offset 0\n");
48     scratch8[0] = 0xa5;
49     // Read back the whole scratch register in one go
50     printf("Should be 0xa5a5a5a5: 0x%08x\n", *scratch32);
51
52     // The IO register ignores the address LSBs [1:0] as well as the transfer
53     // size, so it doesn't matter what byte offset we use
54     printf("\nWriting 8 bit value at offset 1\n");
55     scratch8[1] = 0x3c;
56     printf("Should be 0x3c3c3c3c: 0x%08x\n", *scratch32);
57
58     // Halfword writes are also replicated across the write data bus
59     printf("\nWriting 16 bit value at offset 0\n");
60     scratch16[0] = 0xf00d;
61     printf("Should be 0xf00df00d: 0x%08x\n", *scratch32);
62 }
```

2.1.5. List of Registers

The Bus Fabric registers start at a base address of `0x40030000` (defined as `BUSCTRL_BASE` in SDK).

寄存器的内容。

在不使用读-改-写序列的情况下，更新IO寄存器部分的最佳方案是RP2040上的原子设置/清除/XOR操作（参见第2.1.2节）。请注意，这比字节或半字写操作更灵活，因为任何字段组合都可以在一次操作中更新。

在执行8位或16位写操作（例如Cortex-M0+上的 `strb` 指令）时，IO寄存器将采样完整的32位写入数据总线。RP2040上的 Cortex-M0+和DMA总是会在总线上复制窄宽数据：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/system/narrow_io_write/narrow_io_write.c 第19至62行

```

19 int main() {
20     stdio_init_all();
21
22     // 我们将使用WATCHDOG_SCRATCH0作为方便的32位读写寄存器
23     // 可向其赋予任意值
24     io_rw_32 *scratch32 = &watchdog_hw->scratch[0];
25     // 将scratch寄存器别名为偏移+0x0和+0x2处的两个半字
26     volatile uint16_t *scratch16 = (volatile uint16_t *) scratch32;
27     // 将暂存寄存器别名为偏移量+0x0、+0x1、+0x2、+0x3处的四个字节：
28     volatile uint8_t *scratch8 = (volatile uint8_t *) scratch32;
29
30     // 演示我们可以像正常操作一样读写暂存寄存器：
31     printf("正在写入32位数值\n");
32     *scratch32 = 0xdeadbeef;
33     printf("应为0xdeadbeef: 0x%08x\n", *scratch32);
34
35     // 我们可以正常进行窄宽度读取——IO寄存器将此视为一次32位
36     // 读取，处理器/DMA将根据传输大小和地址最低有效位选择正确的字节通道
37     // on transfer size and address LSBs
38 printf("\n逐字节读取返回\n");
39     // 小端序！
40     printf("应为 ef be ad de: %02x ", scratch8[0]);
41     printf("%02x ", scratch8[1]);
42     printf("%02x ", scratch8[2]);
43     printf("%02x\n", scratch8[3]);
44
45     // 字节写入在32位总线上复制四次，IO
46     // 寄存器通常采样整个写入总线。
47     printf("\n在偏移量0处写入8位值0xa5\n");
48     scratch8[0] = 0xa5;
49     // 一次性读取整个scratch寄存器
50     printf("应为0xa5a5a5a5: 0x%08x\n", *scratch32);
51
52     // IO寄存器忽略地址最低有效位[1:0]及传输
53     // 大小，因此字节偏移无关紧要
54     printf("\n在偏移量1处写入8位值\n");
55     scratch8[1] = 0x3c;
56     printf("应为 0x3c3c3c3c: 0x%08x\n", *scratch32);
57
58     // 半字写入操作也在写数据总线上进行复制
59     printf("\n在偏移量0处写入16位值\n");
60     scratch16[0] = 0xf00d;
61     printf("应为 0xf00df00d: 0x%08x\n", *scratch32);
62 }
```

2.1.5. 寄存器列表

总线结构寄存器的基址起始于 `0x40030000`（在 SDK 中定义为 `BUSCTRL_BASE`）。

Table 4. List of
BUSCTRL registers

Offset	Name	Info
0x00	BUS_PRIORITY	Set the priority of each master for bus arbitration.
0x04	BUS_PRIORITY_ACK	Bus priority acknowledge
0x08	PERFCTR0	Bus fabric performance counter 0
0x0c	PERFSEL0	Bus fabric performance event select for PERFCTR0
0x10	PERFCTR1	Bus fabric performance counter 1
0x14	PERFSEL1	Bus fabric performance event select for PERFCTR1
0x18	PERFCTR2	Bus fabric performance counter 2
0x1c	PERFSEL2	Bus fabric performance event select for PERFCTR2
0x20	PERFCTR3	Bus fabric performance counter 3
0x24	PERFSEL3	Bus fabric performance event select for PERFCTR3

BUSCTRL: BUS_PRIORITY Register

Offset: 0x00

Description

Set the priority of each master for bus arbitration.

Table 5.
BUS_PRIORITY
Register

Bits	Description	Type	Reset
31:13	Reserved.	-	-
12	DMA_W : 0 - low priority, 1 - high priority	RW	0x0
11:9	Reserved.	-	-
8	DMA_R : 0 - low priority, 1 - high priority	RW	0x0
7:5	Reserved.	-	-
4	PROC1 : 0 - low priority, 1 - high priority	RW	0x0
3:1	Reserved.	-	-
0	PROC0 : 0 - low priority, 1 - high priority	RW	0x0

BUSCTRL: BUS_PRIORITY_ACK Register

Offset: 0x04

Description

Bus priority acknowledge

Table 6.
BUS_PRIORITY_ACK
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Goes to 1 once all arbiters have registered the new global priority levels. Arbiters update their local priority when servicing a new nonsequential access. In normal circumstances this will happen almost immediately.	RO	0x0

BUSCTRL: PERFCTR0 Register

Offset: 0x08

表 4. BUSCTRL 寄存器列表

偏移量	名称	说明
0x00	BUS_PRIORITY	设置每个主设备的总线仲裁优先级。
0x04	BUS_PRIORITY_ACK	总线优先级确认
0x08	PERFCTR0	总线结构性能计数器 0
0x0c	PERFSEL0	PERFCTR0 的总线结构性能事件选择
0x10	PERFCTR1	总线结构性能计数器 1
0x14	PERFSEL1	PERFCTR1 的总线结构性能事件选择
0x18	PERFCTR2	总线结构性能计数器 2
0x1c	PERFSEL2	PERFCTR2 的总线结构性能事件选择
0x20	PERFCTR3	总线结构性能计数器 3
0x24	PERFSEL3	PERFCTR3 的总线结构性能事件选择

BUSCTRL: BUS_PRIORITY 寄存器

偏移: 0x00

描述

设置每个主设备的总线仲裁优先级。

表 5. BUS_PRIORITY 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	DMA_W : 0 - 低优先级, 1 - 高优先级	读写	0x0
11:9	保留。	-	-
8	DMA_R : 0 - 低优先级, 1 - 高优先级	读写	0x0
7:5	保留。	-	-
4	PROC1 : 0 - 低优先级, 1 - 高优先级	读写	0x0
3:1	保留。	-	-
0	PROC0 : 0 - 低优先级, 1 - 高优先级	读写	0x0

BUSCTRL: BUS_PRIORITY_ACK 寄存器

偏移: 0x04

描述

总线优先级确认

表 6. BUS_PRIORITY_ACK 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	所有仲裁器注册新的全局优先级后，值变为 1。 仲裁器在处理中断的非连续访问时更新其本地优先级。 在正常情况下，此过程几乎立即发生。	只读	0x0

BUSCTRL: PERFCTR0 寄存器

偏移: 0x08

Description

Bus fabric performance counter 0

Table 7. PERFCTR0 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Busfabric saturating performance counter 0 Count some event signal from the busfabric arbiters. Write any value to clear. Select an event to count using PERFSEL0	WC	0x000000

BUSCTRL: PERFSEL0 Register

Offset: 0x0c

Description

Bus fabric performance event select for PERFCTR0

Table 8. PERFSEL0 Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	Select an event for PERFCTR0. Count either contested accesses, or all accesses, on a downstream port of the main crossbar.	RW	0x1f
	Enumerated values:		
0x00	→ APB_CONTESTED		
0x01	→ APB		
0x02	→ FASTPERI_CONTESTED		
0x03	→ FASTPERI		
0x04	→ SRAM5_CONTESTED		
0x05	→ SRAM5		
0x06	→ SRAM4_CONTESTED		
0x07	→ SRAM4		
0x08	→ SRAM3_CONTESTED		
0x09	→ SRAM3		
0xa	→ SRAM2_CONTESTED		
0xb	→ SRAM2		
0xc	→ SRAM1_CONTESTED		
0xd	→ SRAM1		
0xe	→ SRAM0_CONTESTED		
0xf	→ SRAM0		
0x10	→ XIP_MAIN_CONTESTED		
0x11	→ XIP_MAIN		
0x12	→ ROM_CONTESTED		
0x13	→ ROM		

BUSCTRL: PERFCTR1 Register

描述

总线结构性能计数器 0

表 7. PERFCTR0
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构性能计数器 0 对总线结构仲裁器的某些事件信号进行计数。 写入任意值以清除。通过 PERFSEL0 选择欲计数的事件	WC	0x000000

BUSCTRL: PERFSEL0 寄存器

偏移: 0x0c

说明

PERFCTR0 的总线结构性能事件选择

表 8. PERFSEL0
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR0 选择事件。可计数主交叉开关下游端口上的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

BUSCTRL: PERFCTR1 寄存器

Offset: 0x10**Description**

Bus fabric performance counter 1

Table 9. PERFCTR1 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Busfabric saturating performance counter 1 Count some event signal from the busfabric arbiters. Write any value to clear. Select an event to count using PERFSEL1	WC	0x000000

BUSCTRL: PERFSEL1 Register**Offset:** 0x14**Description**

Bus fabric performance event select for PERFCTR1

Table 10. PERFSEL1 Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	Select an event for PERFCTR1. Count either contested accesses, or all accesses, on a downstream port of the main crossbar.	RW	0x1f
	Enumerated values:		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

偏移: 0x10

说明

总线结构性能计数器 1

表9. PERFCTR1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器1，用于计数来自总线结构仲裁器的特定事件信号。 写入任意值以清除。使用 PERFSEL1 选择要计数的事件	WC	0x000000

BUSCTRL：PERFSEL1 寄存器

偏移: 0x14

说明

PERFCTR1 的总线结构性能事件选择

表10. PERFSEL1
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR1 选择一个事件。计数主交叉开关下游端口的竞争访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

BUSCTRL: PERFCTR2 Register

Offset: 0x18

Description

Bus fabric performance counter 2

Table 11. PERFCTR2 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Busfabric saturating performance counter 2 Count some event signal from the busfabric arbiters. Write any value to clear. Select an event to count using PERFSEL2	WC	0x000000

BUSCTRL: PERFSEL2 Register

Offset: 0x1c

Description

Bus fabric performance event select for PERFCTR2

Table 12. PERFSEL2 Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	Select an event for PERFCTR2. Count either contested accesses, or all accesses, on a downstream port of the main crossbar.	RW	0x1f
	Enumerated values:		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		

BUSCTRL：PERFCTR2 寄存器

偏移：0x18

说明

总线结构性能计数器 2

表11. PERFCTR2 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器2 对总线结构仲裁器的某些事件信号进行计数。 写入任意值以清除。使用 PERFSEL2 选择要计数的事件	WC	0x000000

BUSCTRL：PERFSEL2 寄存器

偏移：0x1c

说明

PERFCTR2 的总线结构性能事件选择

表12. PERFSEL2 寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR2 选择一个事件，统计主交叉开关下游端口的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		

Bits	Description	Type	Reset
	0x13 → ROM		

BUSCTRL: PERFCTR3 Register

Offset: 0x20

Description

Bus fabric performance counter 3

Table 13. PERFCTR3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Busfabric saturating performance counter 3 Count some event signal from the busfabric arbiters. Write any value to clear. Select an event to count using PERFSEL3	WC	0x000000

BUSCTRL: PERFSEL3 Register

Offset: 0x24

Description

Bus fabric performance event select for PERFCTR3

Table 14. PERFSEL3 Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	Select an event for PERFCTR3. Count either contested accesses, or all accesses, on a downstream port of the main crossbar.	RW	0x1f
	Enumerated values:		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		

位	描述	类型	复位值
	0x13 → ROM		

BUSCTRL：PERFCTR3 寄存器

偏移: 0x20

说明

总线结构性能计数器 3

表13. PERFCTR3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	总线结构饱和性能计数器3，统计来自总线结构仲裁器的某些事件信号。 写入任意值以清除。使用 PERFSEL3 选择统计的事件。	WC	0x000000

BUSCTRL：PERFSEL3 寄存器

偏移: 0x24

说明

PERFCTR3的总线结构性能事件选择

表14. PERFSEL3 寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	为 PERFCTR3 选择一个事件，统计主交叉开关下游端口的争用访问或所有访问。	读写	0x1f
	枚举值：		
	0x00 → APB_CONTESTED		
	0x01 → APB		
	0x02 → FASTPERI_CONTESTED		
	0x03 → FASTPERI		
	0x04 → SRAM5_CONTESTED		
	0x05 → SRAM5		
	0x06 → SRAM4_CONTESTED		
	0x07 → SRAM4		
	0x08 → SRAM3_CONTESTED		
	0x09 → SRAM3		
	0x0a → SRAM2_CONTESTED		
	0x0b → SRAM2		
	0x0c → SRAM1_CONTESTED		
	0x0d → SRAM1		
	0x0e → SRAM0_CONTESTED		
	0x0f → SRAM0		
	0x10 → XIP_MAIN_CONTESTED		

Bits	Description	Type	Reset
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

2.2. Address Map

The address map for the device is split in to sections as shown in [Table 15](#). Details are shown in the following sections. Unmapped address ranges raise a bus error when accessed.

2.2.1. Summary

Table 15. Address Map Summary

ROM	0x00000000
XIP	0x10000000
SRAM	0x20000000
APB Peripherals	0x40000000
AHB-Lite Peripherals	0x50000000
IOPORT Registers	0xd0000000
Cortex-M0+ internal registers	0xe0000000

2.2.2. Detail

ROM:

ROM_BASE	0x00000000
----------	------------

XIP:

XIP_BASE	0x10000000
XIP_NOALLOC_BASE	0x11000000
XIP_NOCACHE_BASE	0x12000000
XIP_NOCACHE_NOALLOC_BASE	0x13000000
XIP_CTRL_BASE	0x14000000
XIP_SRAM_BASE	0x15000000
XIP_SRAM_END	0x15004000
XIP_SSI_BASE	0x18000000

SRAM. SRAM0-3 striped:

SRAM_BASE	0x20000000
SRAM_STRIPED_BASE	0x20000000

位	描述	类型	复位值
	0x11 → XIP_MAIN		
	0x12 → ROM_CONTESTED		
	0x13 → ROM		

2.2. 地址映射

设备的地址映射划分为若干部分，如表15所示。详细内容见下文章节。
访问未映射的地址范围时将引发总线错误。

2.2.1. 概述

表15. 地址映射
摘要

ROM	0x00000000
XIP	0x10000000
SRAM	0x20000000
APB 外设	0x40000000
AHB-Lite 外设	0x50000000
IOPORT 寄存器	0xd0000000
Cortex-M0+ 内部寄存器	0xe0000000

2.2.2. 详细说明

只读存储器：

ROM_BASE	0x00000000
----------	------------

XIP:

XIP_BASE	0x10000000
XIP_NOALLOC_BASE	0x11000000
XIP_NOCACHE_BASE	0x12000000
XIP_NOCACHE_NOALLOC_BASE	0x13000000
XIP_CTRL_BASE	0x14000000
XIP_SRAM_BASE	0x15000000
XIP_SRAM_END	0x15004000
XIP_SSI_BASE	0x18000000

SRAM, SRAM0-3 交错：

SRAM_BASE	0x20000000
SRAM_STRIPED_BASE	0x20000000

SRAM_STRIPED_END	0x20040000
------------------	------------

SRAM 4-5 are always non-striped:

SRAM4_BASE	0x20040000
SRAM5_BASE	0x20041000
SRAM_END	0x20042000

Non-striped aliases of SRAM0-3:

SRAM0_BASE	0x21000000
SRAM1_BASE	0x21010000
SRAM2_BASE	0x21020000
SRAM3_BASE	0x21030000

APB Peripherals:

SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40004000
CLOCKS_BASE	0x40008000
RESETS_BASE	0x4000c000
PSM_BASE	0x40010000
IO_BANK0_BASE	0x40014000
IO_QSPI_BASE	0x40018000
PADS_BANK0_BASE	0x4001c000
PADS_QSPI_BASE	0x40020000
XOSC_BASE	0x40024000
PLL_SYS_BASE	0x40028000
PLL_USB_BASE	0x4002c000
BUSCTRL_BASE	0x40030000
UART0_BASE	0x40034000
UART1_BASE	0x40038000
SPI0_BASE	0x4003c000
SPI1_BASE	0x40040000
I2C0_BASE	0x40044000
I2C1_BASE	0x40048000
ADC_BASE	0x4004c000
PWM_BASE	0x40050000
TIMER_BASE	0x40054000
WATCHDOG_BASE	0x40058000
RTC_BASE	0x4005c000

SRAM_STRIPED_END	0x20040000
------------------	------------

SRAM4-5 始终为非交错:

SRAM4_BASE	0x20040000
SRAM5_BASE	0x20041000
SRAM_END	0x20042000

SRAM0-3 的非交错别名:

SRAM0_BASE	0x21000000
SRAM1_BASE	0x21010000
SRAM2_BASE	0x21020000
SRAM3_BASE	0x21030000

APB 外设:

SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40004000
CLOCKS_BASE	0x40008000
RESETS_BASE	0x4000c000
PSM_BASE	0x40010000
IO_BANK0_BASE	0x40014000
IO_QSPI_BASE	0x40018000
PADS_BANK0_BASE	0x4001c000
PADS_QSPI_BASE	0x40020000
XOSC_BASE	0x40024000
PLL_SYS_BASE	0x40028000
PLL_USB_BASE	0x4002c000
BUSCTRL_BASE	0x40030000
UART0_BASE	0x40034000
UART1_BASE	0x40038000
SPI0_BASE	0x4003c000
SPI1_BASE	0x40040000
I2C0_BASE	0x40044000
I2C1_BASE	0x40048000
ADC_BASE	0x4004c000
PWM_BASE	0x40050000
TIMER_BASE	0x40054000
WATCHDOG_BASE	0x40058000
RTC_BASE	0x4005c000

ROSC_BASE	0x40060000
VREG_AND_CHIP_RESET_BASE	0x40064000
TBMAN_BASE	0x4006c000

AHB-Lite peripherals:

DMA_BASE	0x50000000
----------	------------

USB has a DPRAM at its base followed by registers:

USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000

Remaining AHB-Lite peripherals:

PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
XIP_AUX_BASE	0x50400000

IOPORT Peripherals:

SIO_BASE	0xd0000000
----------	------------

Cortex-M0+ Internal Peripherals:

PPB_BASE	0xe0000000
----------	------------

2.3. Processor subsystem

The RP2040 processor subsystem consists of two Arm Cortex-M0+ processors – each with its standard internal Arm CPU peripherals – alongside external peripherals for GPIO access and inter-core communication. Details of the Arm Cortex-M0+ processors, including the specific feature configuration used on RP2040, can be found in [Section 2.4](#).

ROSC_BASE	0x40060000
VREG_AND_CHIP_RESET_BASE	0x40064000
TBMAN_BASE	0x4006c000

AHB-Lite 外设：

DMA_BASE	0x50000000
----------	------------

USB 基址处设有 DPRAM，紧接其后为寄存器：

USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000

其余 AHB-Lite 外设：

PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
XIP_AUX_BASE	0x50400000

IOPORT 外设：

SIO_BASE	0xd0000000
----------	------------

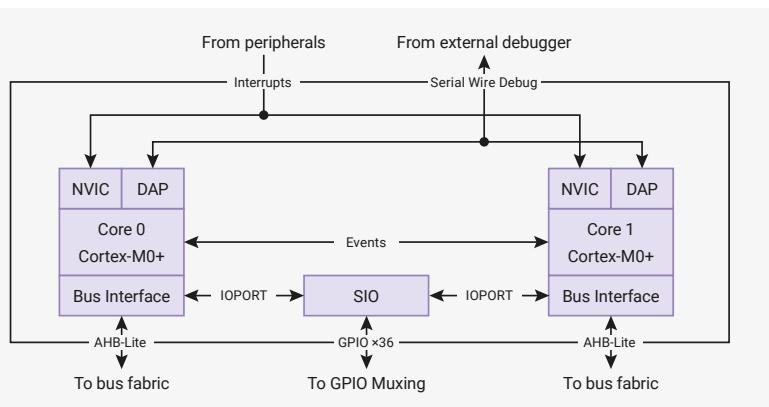
Cortex-M0+ 内部外设：

PPB_BASE	0xe0000000
----------	------------

2.3. 处理器子系统

RP2040 处理器子系统由两个 Arm Cortex-M0+ 处理器组成，每个处理器均配备标准的内部 Arm CPU 外设，以及用于 GPIO 访问和核间通信的外部外设。有关 Arm Cortex-M0+ 处理器的详细信息，包括 RP2040 上使用的特定功能配置，请参见第 2.4 节。

Figure 6. Two Cortex-M0+ processors, each with a dedicated 32-bit AHB-Lite bus port, for code fetch, loads and stores. The SIO is connected to the single-cycle IOPORT bus of each processor, and provides GPIO access, two-way communications, and other core-local peripherals. Both processors can be debugged via a single multi-drop Serial Wire Debug bus. 26 interrupts (plus NMI) are routed to the NVIC and WIC on each processor.



NOTE

The terms *core0* and *core1*, *proc0* and *proc1* are used interchangeably in RP2040's registers and documentation to refer to processor 0, and processor 1 respectively.

The processors use a number of interfaces to communicate with the rest of the system:

- Each processor uses its own independent 32-bit AHB-Lite bus to access memory and memory-mapped peripherals (more detail in [Section 2.1](#))
- The single-cycle IO block provides high-speed, deterministic access to GPIOs via each processor's IOPORT
- 26 system-level interrupts are routed to both processors
- A multi-drop Serial Wire Debug bus provides debug access to both processors from an external debug host

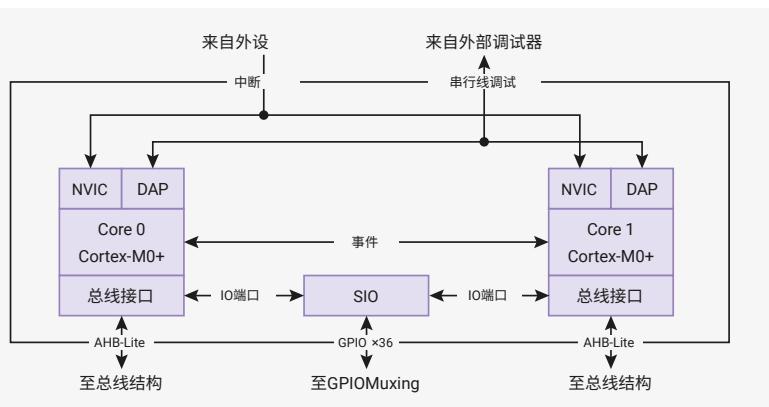
2.3.1. SIO

The Single-cycle IO block (SIO) contains several peripherals that require low-latency, deterministic access from the processors. It is accessed via each processor's IOPORT: this is an auxiliary bus port on the Cortex-M0+ which can perform rapid 32-bit reads and writes. The SIO has a dedicated bus interface for each processor's IOPORT, as shown in [Figure 7](#). Processors access their IOPORT with normal load and store instructions, directed to the special IOPORT address segment, `0xd0000000…0xffffffff`. The SIO appears as memory-mapped hardware within the IOPORT space.

NOTE

The SIO is not connected to the main system bus due to its tight timing requirements. It can only be accessed by the processors, or by the debugger via the processor debug ports.

图6. 两个Cortex-M0+处理器，每个均配备专用的32位AHB-Lite总线端口，用于代码获取、加载及存储。SIO连接至每个处理器的单周期IOPORT总线，提供GPIO访问、双向通信及其他核心本地外设功能。两个处理器均可通过单一的多点Serial Wire Debug总线进行调试。26个中断（加上NMI）分别路由至每个处理器的NVIC和WIC。



注意

术语 core0 和 core1、proc0 和 proc1 在 RP2040 的寄存器及文档中交替使用，分别指代处理器0和处理器1。

处理器使用多种接口与系统的其他部分进行通信：

- 每个处理器使用其独立的32位AHB-Lite总线访问内存和内存映射外设。
(详见第2.1节)
- 单周期IO模块通过每个处理器的IOPORT提供对GPIO的高速、确定性访问。
- 26个系统级中断被路由至两个处理器。
- 多点Serial Wire Debug总线为外部调试主机提供对两个处理器的调试访问。

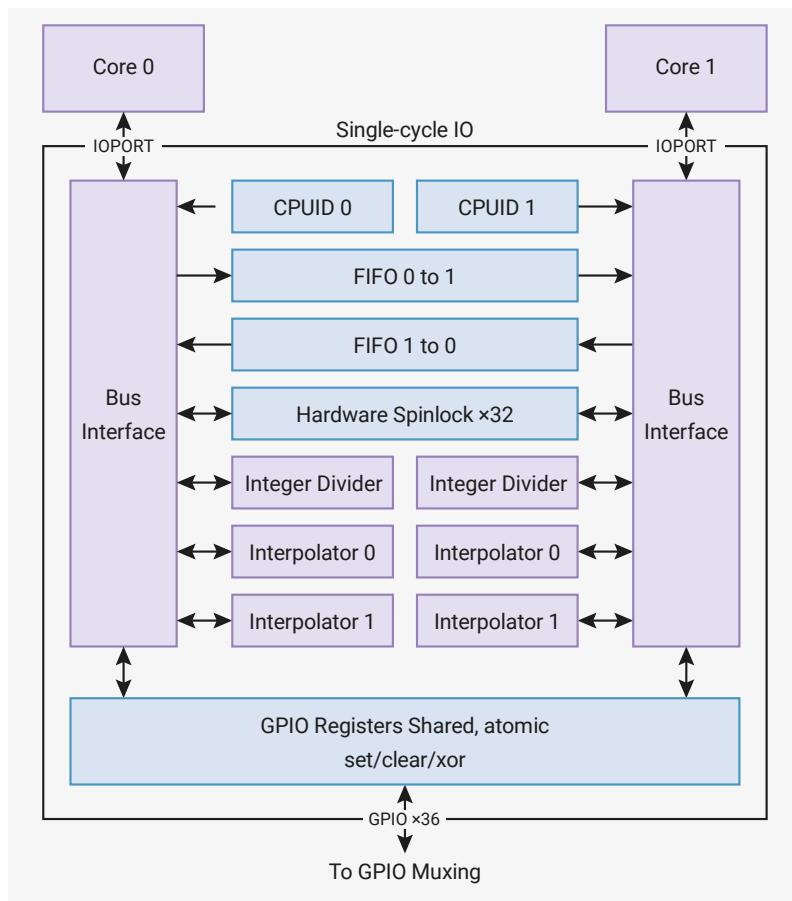
2.3.1. SIO

单周期IO模块（SIO）包含多个需要低延迟且确定性访问的外设。它通过每个处理器的IOPORT访问：这是 Cortex-M0+ 上的辅助总线端口，能够执行快速的 32 位读写。SIO 为每个处理器的 IOPORT 提供了专用的总线接口，如图7所示。处理器通过指向特殊 IOPORT 地址段 `0xd` 的正常加载和存储指令访问其 IOPORT。`0000000…0xffffffff`。SIO 作为内存映射硬件，在 IOPORT 地址空间内呈现。

注意

鉴于其严格的时序要求，SIO 并未连接至主系统总线。仅限处理器访问，或通过处理器调试端口由调试器访问。

Figure 7. The single-cycle I/O block contains memory-mapped hardware which the processors must be able to access quickly. The FIFOs and spinlocks support message passing and synchronisation between the two cores. The shared GPIO registers provide fast and concurrency-safe direct access to GPIO-capable pins. Some core-local arithmetic hardware can be used to accelerate common tasks on the processors.



All IOPORT reads and writes (and therefore all SIO accesses) take place in exactly one cycle, unlike the main AHB-Lite system bus, where the Cortex-M0+ requires two cycles for a load or store, and may have to wait longer due to contention from other system bus masters. This is vital for interfaces such as GPIO, which have tight timing requirements.

SIO registers are mapped to word-aligned addresses in the range `0xd0000000…0xd000017c`. The remainder of the IOPORT space is reserved for future use.

The SIO peripherals are described in more detail in the following sections.

2.3.1.1. CPUID

The register **CPUID** is the first register in the IOPORT space. Core 0 reads a value of 0 when accessing this address, and core 1 reads a value of 1. This is a convenient method for software to determine on which core it is running. This is checked during the initial boot sequence: both cores start running simultaneously, core 1 goes into a deep sleep state, and core 0 continues with the main boot sequence.

! IMPORTANT

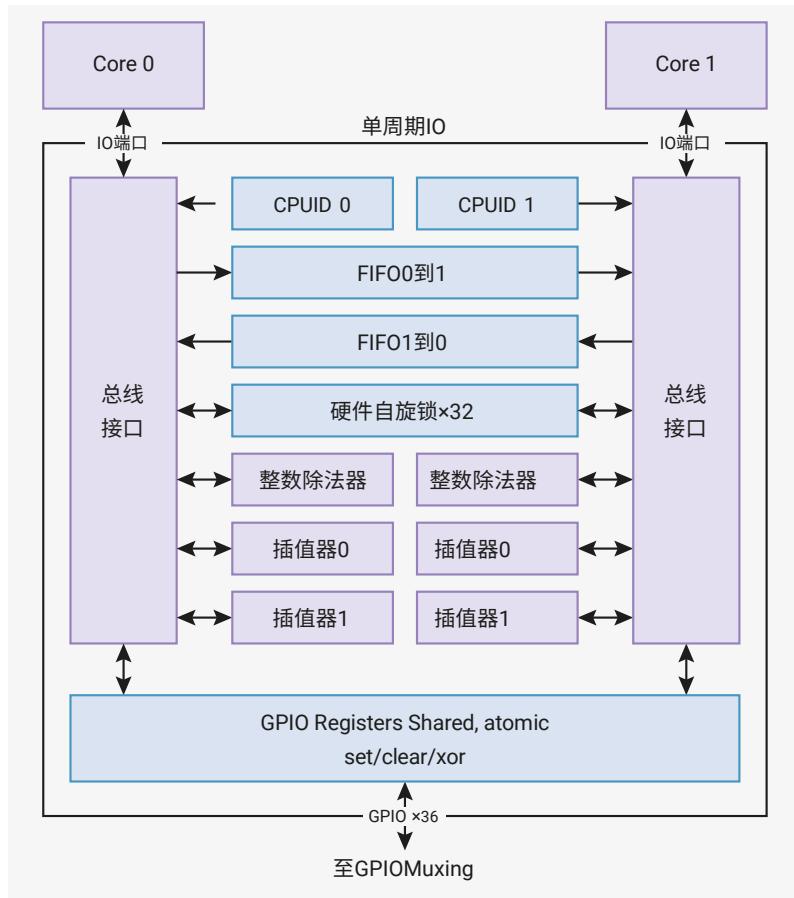
CPUID should not be confused with the Cortex-M0+ CPUID register ([Section 2.4.4.1.1](#)) on each processor's internal Private Peripheral Bus, which lists the processor's part number and version.

2.3.1.2. GPIO Control

The processors have access to GPIO registers for fast and direct control of pins with GPIO functionality. There are two identical sets of registers:

图7. 单周期IO模块包含处理器必须能够快速访问的内存映射硬件。FIFO和自旋锁支持两个核心之间的消息传递与同步。共享GPIO寄存器提供对支持GPIO的引脚的快速且并发安全的直接访问。

部分内核本地算术硬件可用于加速处理器上的常见任务。



所有IOPORT的读写操作（因此所有SIO访问）均在恰好一个周期内完成，这与Cortex-M0+的主AHB-Lite系统总线不同，后者的加载或存储操作需要两个周期，且可能因其他系统总线主控器的争用而等待更久。这对诸如GPIO等具有严格时序要求的接口至关重要。

SIO寄存器映射至范围为`0xd`的字对齐地址`00000000…0xd000017c`。IOPORT空间的其余部分保留以供将来使用。

SIO外设将在下列章节中作更详细的描述。

2.3.1.1. CPUID

寄存器CPUID是IOPORT空间中的第一个寄存器。核心0访问此地址时读取的值为0，核心1读取的值为1。这是软件确定其运行核心的便捷方法。此项检查在初始引导序列中进行：两个核心同时启动，核心1进入深度睡眠状态，核心0继续执行主引导序列。

！重要

CPUID不应与每个处理器内部私有外设总线上的Cortex-M0+ CPUID寄存器（第2.4.4.1.1节）混淆，后者列示了处理器的部件号及版本。

2.3.1.2. GPIO控制

处理器可通过GPIO寄存器对具GPIO功能的引脚进行快速且直接的控制。存在两套相同的寄存器：

- `GPIO_x` for direct control of IO bank 0 (user GPIOs 0 to 29, starting at the LSB)
- `GPIO_HI_x` for direct control of the QSPI IO bank (in the order SCLK, SSn, SD0, SD1, SD2, SD3, starting at the LSB)

NOTE

To drive a pin with the SIO's GPIO registers, the GPIO multiplexer for this pin must first be configured to select the SIO GPIO function. See [Table 279](#).

These GPIO registers are *shared* between the two cores, and both cores can access them simultaneously. There are three registers for each bank:

- Output registers, `GPIO_OUT` and `GPIO_HI_OUT`, are used to set the output level of the GPIO (1/0 for high/low)
- Output enable registers, `GPIO_OE` and `GPIO_HI_OE`, are used to enable the output driver. 0 for high-impedance, 1 for drive high/low based on `GPIO_OUT` and `GPIO_HI_OUT`.
- Input registers, `GPIO_IN` and `GPIO_HI_IN`, allow the processor to sample the current state of the GPIOs

Reading `GPIO_IN` returns all 30 GPIO values (or 6 for `GPIO_HI_IN`) in a single read. Software can then mask out individual pins it is interested in.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware_gpio.h Lines 859 - 869

```

859 static inline bool gpio_get(uint gpio) {
860 #ifdef NUM_BANK0_GPIOS <= 32
861     return sio_hw->gpio_in & (1u << gpio);
862 #else
863     if (gpio < 32) {
864         return sio_hw->gpio_in & (1u << gpio);
865     } else {
866         return sio_hw->gpio_hi_in & (1u << (gpio - 32));
867     }
868 #endif
869 }
```

The `OUT` and `OE` registers also have atomic SET, CLR, and XOR aliases, which allows software to update a subset of the pins in one operation. This is vital not only for safe parallel GPIO access between the two cores, but also safe concurrent GPIO access in an interrupt handler and foreground code running on one core.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware_gpio.h Lines 908 - 914

```

908 static inline void gpio_set_mask(uint32_t mask) {
909 #ifdef PICO_USE_GPIO_COPROCESSOR
910     gpioc_lo_out_set(mask);
911 #else
912     sio_hw->gpio_set = mask;
913 #endif
914 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware_gpio.h Lines 955 - 961

```

955 static inline void gpio_clr_mask(uint32_t mask) {
956 #ifdef PICO_USE_GPIO_COPROCESSOR
957     gpioc_lo_out_clr(mask);
958 #else
959     sio_hw->gpio_clr = mask;
960 #endif
961 }
```

- `GPIO_x` 用于直接控制IO银行0（用户GPIO 0至29，从最低有效位起）
- `GPIO_HI_x` 用于直接控制QSPI IO银行（按顺序为SCLK、SSn、SD0、SD1、SD2、SD3，从最低有效位起）

注意

要通过SIO的GPIO寄存器驱动引脚，必须首先将该引脚的GPIO多路复用器配置为选择SIO GPIO功能。详见表279。

这些GPIO寄存器在双核之间共享，且两核可同时访问。每个银行包含三个寄存器：

- 输出寄存器`GPIO_OUT`和`GPIO_HI_OUT`用于设置GPIO的输出电平（1表示高电平，0表示低电平）
- 输出使能寄存器`GPIO_OE`和`GPIO_HI_OE`用于使能输出驱动。0表示高阻态，1表示依据`GPIO_OUT`和`GPIO_HI_OUT`驱动高/低电平。
- 输入寄存器`GPIO_IN`和`GPIO_HI_IN`允许处理器采样GPIO的当前状态。

读取`GPIO_IN`一次即可返回全部30个GPIO的值（`GPIO_HI_IN`为6个）。软件随后可屏蔽不感兴趣的引脚，仅保留所需引脚。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h, 第859至869行

```

859 static inline bool gpio_get(uint gpio) {
860 #ifdef NUM_BANK0_GPIOS <= 32
861     return sio_hw->gpio_in & (1u << gpio);
862 #else
863     if (gpio < 32) {
864         return sio_hw->gpio_in & (1u << gpio);
865     } else {
866         return sio_hw->gpio_hi_in & (1u << (gpio - 32));
867     }
868 #endif
869 }
```

`OUT`和`OE`寄存器同样具备原子性的`SET`、`CLR`和`XOR`别名，从而允许软件通过一次操作更新部分引脚。这不仅对于两个核心之间安全的并行GPIO访问至关重要，也保证了中断处理程序与在单核上运行的前台代码对GPIO的安全并发访问。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h 第908至914行

```

908 static inline void gpio_set_mask(uint32_t mask) {
909 #ifdef PICO_USE_GPIO_COPROCESSOR
910     gpioc_lo_out_set(mask);
911 #else
912     sio_hw->gpio_set = mask;
913 #endif
914 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h 第955至961行

```

955 static inline void gpio_clr_mask(uint32_t mask) {
956 #ifdef PICO_USE_GPIO_COPROCESSOR
957     gpioc_lo_out_clr(mask);
958 #else
959     sio_hw->gpio_clr = mask;
960 #endif
961 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware_gpio.h Lines 1145 - 1170

```

1145 static inline void gpio_put(uint gpio, bool value) {
1146 #ifdef PICO_USE_GPIO_COPROCESSOR
1147     gpioc_bit_out_put(gpio, value);
1148 #elif NUM_BANK0_GPIOS <= 32
1149     uint32_t mask = 1ul << gpio;
1150     if (value)
1151         gpio_set_mask(mask);
1152     else
1153         gpio_clr_mask(mask);
1154 #else
1155     uint32_t mask = 1ul << (gpio & 0x1fu);
1156     if (gpio < 32) {
1157         if (value) {
1158             sio_hw->gpio_set = mask;
1159         } else {
1160             sio_hw->gpio_clr = mask;
1161         }
1162     } else {
1163         if (value) {
1164             sio_hw->gpio_hi_set = mask;
1165         } else {
1166             sio_hw->gpio_hi_clr = mask;
1167         }
1168     }
1169 #endif
1170 }
```

If both processors write to an **OUT** or **OE** register (or any of its SET/CLR/XOR aliases) on the same clock cycle, the result is as though core 0 wrote first, and core 1 wrote immediately afterward. For example, if core 0 SETs a bit, and core 1 simultaneously XORs it, the bit will be set to 0, irrespective of its original value.

NOTE

This is a conceptual model for the result that is produced when two cores write to a GPIO register simultaneously. The register does not actually contain this intermediate value at any point. In the previous example, if the pin is initially 0, and core 0 performs a SET while core 1 performs a XOR, the GPIO output remains low without any positive glitch.

2.3.1.3. Hardware Spinlocks

The SIO provides 32 hardware spinlocks, which can be used to manage mutually-exclusive access to shared software resources. Each spinlock is a one-bit flag, mapped to a different address (from **SPINLOCK0** to **SPINLOCK31**). Software interacts with each spinlock with one of the following operations:

- Read: attempt to claim the lock. Read value is nonzero if the lock was successfully claimed, or zero if the lock had already been claimed by a previous read.
- Write (any value): release the lock. The next attempt to claim the lock will be successful.

If both cores try to claim the same lock on the same clock cycle, core 0 succeeds.

Generally software will acquire a lock by repeatedly polling the lock bit ("spinning" on the lock) until it is successfully claimed. This is inefficient if the lock is held for long periods, so generally the spinlocks should be used to protect the short critical sections of higher-level primitives such as mutexes, semaphores and queues.

For debugging purposes, the current state of all 32 spinlocks can be observed via **SPINLOCK_ST**.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/include/hardware/gpio.h, 第1145至1170行

```

1145 static inline void gpio_put(uint gpio, bool value) {
1146 #ifdef PICO_USE_GPIO_COPROCESSOR
1147     gpioc_bit_out_put(gpio, value);
1148 #elif NUM_BANK0_GPIOS <= 32
1149     uint32_t mask = 1ul << gpio;
1150     if (value)
1151         gpio_set_mask(mask);
1152     else
1153         gpio_clr_mask(mask);
1154 #else
1155     uint32_t mask = 1ul << (gpio & 0x1fu);
1156     if (gpio < 32) {
1157         if (value) {
1158             sio_hw->gpio_set = mask;
1159         } else {
1160             sio_hw->gpio_clr = mask;
1161         }
1162     } else {
1163         if (value) {
1164             sio_hw->gpio_hi_set = mask;
1165         } else {
1166             sio_hw->gpio_hi_clr = mask;
1167         }
1168     }
1169 #endif
1170 }
```

如果两个处理器在同一时钟周期内同时写入 OUT 或 OE 寄存器（或其任一SET/CLR/XOR别名），结果相当于核心0先写入，随后核心1紧接着写入。例如，若核心0对某位执行SET操作，而核心1同时对该位执行XOR操作，则该位将被设置为0，且与其原始值无关。

① 注意

这是两个核心同时写入GPIO寄存器时所产生结果的概念模型。

该寄存器在任何时刻实际上并不包含该中间值。在前述示例中，若引脚初值为0，且核心0执行SET而核心1执行XOR，则GPIO输出保持低电平，不产生任何正向毛刺。

2.3.1.3 硬件自旋锁

SIO 提供 32 个硬件自旋锁，用于管理对共享软件资源的互斥访问。每个自旋锁均为一位标志，映射至不同地址（自 SPINLOCK0 至 SPINLOCK31）。软件通过以下操作之一与自旋锁交互：

- 读取：尝试获取锁。读取值非零则表示锁已成功获取，读取值为零则表示锁已被先前的读取操作获取。

- 写入（任意值）：释放锁。下一次尝试获取该锁将成功。

若两个核心在同一时钟周期尝试获取同一锁，则核心0优先成功。

一般情况下，软件通过重复轮询锁位（即“自旋”）直到成功获取锁。若锁长时间被持有，此方法效率较低，故自旋锁通常用于保护互斥锁、信号量和队列等高级原语中的短临界区。

出于调试目的，可以通过 SPINLOCK_ST 查看所有 32 个自旋锁的当前状态。

2.3.1.4. Inter-processor FIFOs (Mailboxes)

The SIO contains two FIFOs for passing data, messages or ordered events between the two cores. Each FIFO is 32 bits wide, and eight entries deep. One of the FIFOs can only be written by core 0, and read by core 1. The other can only be written by core 1, and read by core 0.

Each core writes to its outgoing FIFO by writing to **FIFO_WR**, and reads from its incoming FIFO by reading from **FIFO_RD**. A status register, **FIFO_ST**, provides the following status signals:

- Incoming FIFO contains data (**VLD**)
- Outgoing FIFO has room for more data (**RDY**)
- The incoming FIFO was read from while empty at some point in the past (**ROE**)
- The outgoing FIFO was written to while full at some point in the past (**WOF**)

Writing to the outgoing FIFO while full, or reading from the incoming FIFO while empty, does not affect the FIFO state. The current contents and level of the FIFO is preserved. However, this does represent some loss of data or reception of invalid data by the software accessing the FIFO, so a sticky error flag is raised (**ROE** or **WOF**).

The SIO has a FIFO IRQ output for each core, mapped to system IRQ numbers **15** and **16**. Each IRQ output is the logical OR of the **VLD**, **ROE** and **WOF** bits in that core's **FIFO_ST** register: that is, the IRQ is asserted if any of these three bits is high, and clears again when they are all low. The **ROE** and **WOF** flags are cleared by writing any value to **FIFO_ST**, and the **VLD** flag is cleared by reading data from the FIFO until empty.

If the corresponding interrupt line is enabled in the Cortex-M0+ NVIC, then the processor will take an interrupt each time data appears in its FIFO, or if it has performed some invalid FIFO operation (read on empty, write on full). Typically Core 0 will use IRQ15 and core 1 will use IRQ16. If the IRQs are used the other way round then it is difficult for the core that has been interrupted to correctly identify the reason for the interrupt as the core doesn't have access to the other core's FIFO status register.

NOTE

ROE and **WOF** only become set if software misbehaves in some way. Generally, the interrupt handler will trigger when data appears in the FIFO (raising the **VLD** flag), and the interrupt handler clears the IRQ by reading data from the FIFO until **VLD** goes low once more.

The inter-processor FIFOs and the Cortex-M0+ Event signals are used by the bootrom (Section 2.8) **wait_for_vector** routine, where core 1 remains in a sleep state until it is woken, and provided with its initial stack pointer, entry point and vector table through the FIFO.

2.3.1.5. Integer Divider

The SIO provides one 8-cycle signed/unsigned divide/modulo module to each of the cores. Calculation is started by writing a dividend and divisor to the two argument registers, **DIVIDEND** and **DIVISOR**. The divider calculates the quotient / and remainder % of this division over the next 8 cycles, and on the 9th cycle the results can be read from the two result registers **DIV_QUOTIENT** and **DIV_REMAINDER**. A 'ready' bit in register **DIV_CSR** can be polled to wait for the calculation to complete, or software can insert a fixed 8-cycle delay.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S Lines 12 - 16

```

12 regular_func_with_section hw_divider_divmod_s32
13     ldr r3, =(SIO_BASE)
14     str r0, [r3, #SIO_DIV_SDIVIDEND_OFFSET]
15     str r1, [r3, #SIO_DIV_SDIVISOR_OFFSET]
16     b hw_divider_divmod_return

```

2.3.1.4. 处理器间 FIFO (邮件箱)

SIO 包含两个 FIFO，用于在两个核心之间传递数据、消息或有序事件。每个 FIFO 宽度为 32 位，深度为八个条目。其中一个 FIFO 仅能由核心 0 写入，由核心 1 读取。另一个 FIFO 仅能由核心 1 写入，由核心 0 读取。

每个核心通过写入 FIFO_WR 向其输出 FIFO 写入数据，通过读取 FIFO_RD 读取其输入 FIFO。

状态寄存器 FIFO_ST 提供以下状态信号：

- 输入 FIFO 中含有数据 (VLD)
- 输出 FIFO 有空间可用 (RDY)
- 输入 FIFO 曾在空时被读取 (ROE)
- 输出 FIFO 曾在满时被写入 (WOF)

在发送 FIFO 已满时写入，或者在接收 FIFO 为空时读取，不会改变 FIFO 的状态。

FIFO 的当前内容及其级别状态将被保留。然而，这确实可能导致访问 FIFO 的软件发生数据丢失或接收无效数据，因此将置位一个粘性错误标志 (ROE 或 WOF)。

SIO 为每个核心提供一个 FIFO 中断输出，映射至系统中断号 15 和 16。每个中断输出端是对应核心 FIFO_ST 寄存器中 VLD、RDY 及 WOF 位的逻辑或：当任一位为高电平时，中断信号被置位；当三者均为低电平时，中断信号清除。通过向 FIFO_ST 写入任意数值可以清除 ROE 和 WOF 标志，通过从 FIFO 读取数据直至其为空可以清除 VLD 标志。

若相应中断线在 Cortex-M0+ 的 NVIC 中被使能，则每当其 FIFO 中有数据出现，或发生无效的 FIFO 操作（空读或满写）时，处理器将响应中断。通常，核心 0 将使用 IRQ15，核心 1 将使用 IRQ16。如果 IRQs 使用顺序相反，被中断的核心很难正确识别中断原因，因为该核心无法访问另一个核心的 FIFO 状态寄存器。

注意

只有当软件异常时，ROE 和 WOF 才会被置位。通常，当 FIFO 中出现数据（触发 VLD 标志）时，中断处理程序会被触发，并通过从 FIFO 读取数据直至 VLD 恢复为低电平来清除 IRQ。

启动只读存储器（第 2.8 节）中的 `wait_for_vector` 例程使用了核间 FIFO 和 Cortex-M0+ 事件信号，其中核心 1 保持睡眠状态直至被唤醒，并通过 FIFO 提供其初始堆栈指针、入口点和向量表。

2.3.1.5 整数除法器

SIO 为每个核心提供一套 8 周期的有符号/无符号除法及取模模块。计算通过向两个参数寄存器 DIVIDEND 和 DIVISOR 写入被除数和除数来启动。除法器在接下来的 8 个周期内计算该除法的商 / 和余数 %，第 9 个周期可从两个结果寄存器 DIV_QUOTIENT 和 DIV_REMAINDER 读取结果。可通过轮询寄存器 DIV_CSR 中的“就绪”位等待计算完成，或软件插入固定 8 周期延迟。

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S 第 12 至 16 行

```

12 regular_func_with_section hw_divider_divmod_s32
13     ldr r3, =(SIO_BASE)
14     str r0, [r3, #SIO_DIV_SDIVIDEND_OFFSET]
15     str r1, [r3, #SIO_DIV_SDIVISOR_OFFSET]
16     b hw_divider_divmod_return

```

NOTE

Software is free to perform other non-divider operations during these 8 cycles.

There are two aliases of the operand registers: writing to the signed alias (`DIV_SDIVIDEND` and `DIV_SDIVISOR`) will initiate a signed calculation, and the other (`DIV_UDIVIDEND` and `DIV_UDIVISOR`) will initiate an unsigned calculation.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S Lines 20 - 24

```
20 regular_func_with_section hw_divider_divmod_u32
21     ldr r3, =(SIO_BASE)
22     str r0, [r3, #SIO_DIV_UDIVIDEND_OFFSET]
23     str r1, [r3, #SIO_DIV_UDIVISOR_OFFSET]
24     b hw_divider_divmod_return
```

NOTE

A new calculation begins immediately with every write to an operand register, and a new operand write immediately squashes any calculation currently in progress. For example, when dividing many numbers by the same divisor, only `xDIVISOR` needs to be written, and the signedness of each calculation is determined by whether `SDIVIDEND` or `UDIVIDEND` is written.

To support save and restore on interrupt handler entry/exit (or on e.g. an RTOS context switch), the result registers are also writable. Writing to a result register will cancel any operation in progress at the time. The `DIV_CSR.DIRTY` flag can help make save/restore more efficient: this flag is set when *any* divider register (operand or result) is written to, and cleared when the quotient is read.

NOTE

When enabled, the default divider AEABI support maps C level `/` and `%` to the hardware divider. When building software using the SDK and using the divider directly, it is important to read the quotient register *last*. This ensures the partial divider state will be correctly saved and restored by any interrupt code that uses the divider. You should read the quotient register whether you need the value or not.

The SDK module `pico_divider` https://github.com/raspberrypi/pico-sdk/blob/master/src/common/pico_divider_headers/include/pico/divider.h provides both the *AEABI* implementation needed to hook the C `/` and `%` operators for both 32-bit and 64-bit integer division, as well as some additional C functions that return quotients and remainders at the same time. All of these functions correctly save and restore the hardware divider state (when dirty) so that they can be used in either user or IRQ handler code.

The SDK module `hardware_divider` https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/include/hardware/divider.h provides lower level macros and helper functions for accessing the hardware_divider, but these do not save and restore the hardware divider state (although this header does provide separate functions to do so).

2.3.1.6. Interpolator

Each core is equipped with two *interpolators* (`INTERP0` and `INTERP1`) which can accelerate tasks by combining certain pre-configured operations into a single processor cycle. Intended for cases where the pre-configured operation is repeated many times, this results in code which uses both fewer CPU cycles and fewer CPU registers in the time-critical sections of the code.

The interpolators are used to accelerate audio operations within the SDK, but their flexible configuration makes it possible to optimise many other tasks such as quantization and dithering, table lookup address generation, affine texture mapping, decompression and linear feedback.

注意

软件在这8个周期内可自由执行其他非除法器操作。

操作数寄存器有两种别名：写入有符号别名（DIV_SDIVIDEND 与 DIV_SDIVISOR）将启动有符号计算，写入无符号别名（DIV_UDIVIDEND 与 DIV_UDIVISOR）将启动无符号计算。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/divider.S 第20至24行

```
20 regular_func_with_section hw_divider_divmod_u32
21     ldr r3, =(SIO_BASE)
22     str r0, [r3, #SIO_DIV_UDIVIDEND_OFFSET]
23     str r1, [r3, #SIO_DIV_UDIVISOR_OFFSET]
24     b hw_divider_divmod_return
```

注意

每次对操作数寄存器写入时，都会立即开始新的计算，且新的操作数写入会立即终止当前正在进行的任何计算。例如，在用相同除数除多个数时，仅需写入 xDIVISOR，且每次计算的符号由是否写入 SDIVIDEND 或 UDIVIDEND 决定。

为支持中断处理程序入口/出口（或如 RTOS 上下文切换）时的保存与恢复，结果寄存器同样可写。写入结果寄存器将取消当时正在进行的任何操作。DIV_CSR.DIRTY 标志有助于提升保存/恢复效率：当任何除法器寄存器（操作数或结果）被写入时，该标志置位；读取时则清除该标志。

注意

启用后，默认除法器 AEABI 支持将 C 语言中的除法符号 / 和取余符号 % 映射至硬件除法器。使用 SDK 构建软件并直接使用除法器时，务必最后读取商寄存器。这保证了部分除法器状态能够被任何使用除法器的中断代码正确保存和恢复。无论是否需要该值，均应读取商寄存器。

SDK模块https://github.com/raspberrypi/pico-sdk/blob/master/src/common/pico_divider_headers/include/pico/divider.h

 提供了对32位和64位整数除法中钩载C语言/和 % 运算符所需的AEABI实现，以及一些能够同时返回商和余数的附加C函数。所有这些函数在硬件除法器状态“脏”时，均能正确保存并恢复该状态，以便可用于用户代码或中断处理程序代码中。

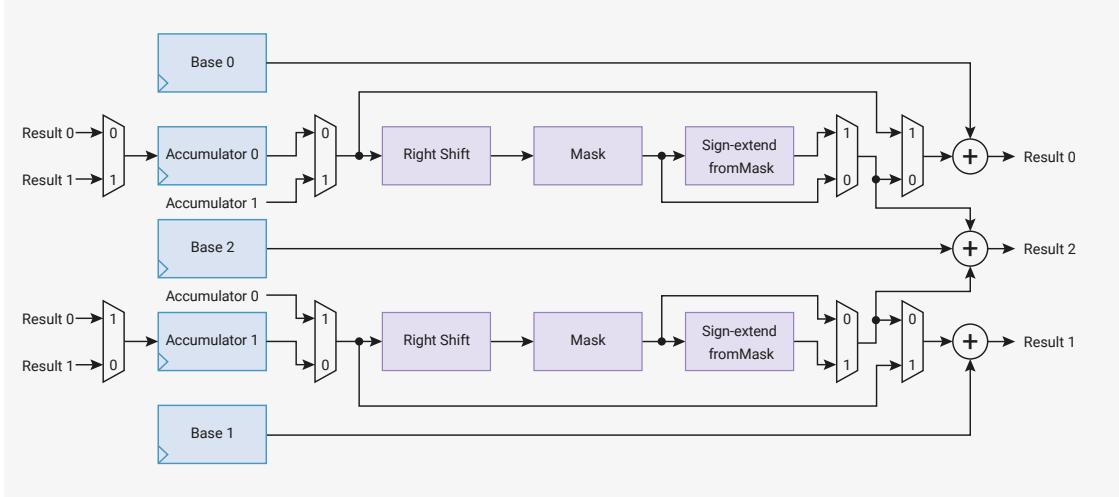
SDK模块hardware_dividerhttps://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_divider/include/hardware/divider.h 提供了访问硬件除法器的底层宏和辅助函数，但这些函数不保存或恢复硬件除法器状态（尽管该头文件确实提供了单独的函数来实现此功能）。

2.3.1.6. 插值器

每个核心配备了两个插值器（INTERP0 和 INTERP1），能够通过将特定预配置操作合并为单个处理器周期以加速任务。适用于预配置操作重复多次的场景，从而使代码在时间关键段使用更少的CPU周期和寄存器。

插值器用于加速SDK中的音频操作，其灵活的配置亦使得量化与抖动、查表地址生成、仿射纹理映射、解压缩及线性反馈等多种任务得到优化。

Figure 8. An interpolator. The two accumulator registers and three base registers have single-cycle read/write access from the processor. The interpolator is organised into two lanes, which perform masking, shifting and sign-extension operations on the two accumulators. This produces three possible results, by adding the intermediate shift/mask values to the three base registers. From left to right, the multiplexers on each lane are controlled by the following flags in the **CTRL** registers: **CROSS_RESULT**, **CROSS_INPUT**, **SIGNED**, **ADD_RAW**.



The processor can write or read any interpolator register in one cycle, and the results are ready on the next cycle. The processor can also perform an addition on one of the two accumulators **ACCUM0** or **ACCUM1** by writing to the corresponding **ACCUMx_ADD** register.

The three results are available in the read-only locations **PEEK0**, **PEEK1**, **PEEK2**. Reading from these locations does not change the state of the interpolator. The results are also aliased at the locations **POP0**, **POP1**, **POP2**; reading from a **POPx** alias returns the same result as the corresponding **PEEKx**, and simultaneously writes back the lane results to the accumulators. This can be used to advance the state of interpolator each time a result is read.

Additionally the interpolator supports simple fractional blending between two values as well as clamping values such that they lie within a given range.

The following example shows a trivial example of popping a lane result to produce simple iterative feedback.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 11 - 23

```

11 void times_table() {
12     puts("9 times table:");
13
14     // Initialise lane 0 on interp0 on this core
15     interp_config cfg = interp_default_config();
16     interp_set_config(interp0, 0, &cfg);
17
18     interp0->accum[0] = 0;
19     interp0->base[0] = 9;
20
21     for (int i = 0; i < 10; ++i)
22         printf("%d\n", interp0->pop[0]);
23 }

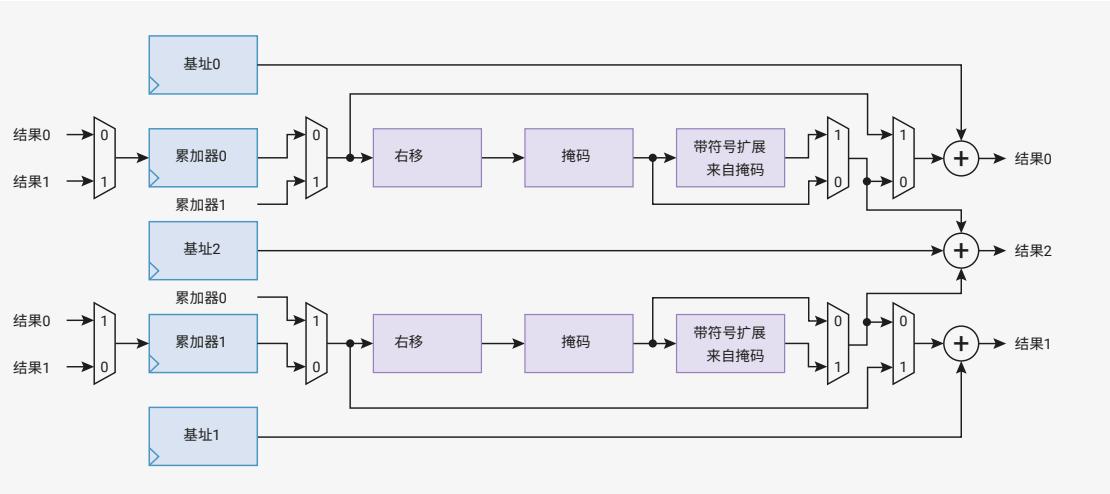
```

NOTE

By sheer coincidence, the interpolators are extremely well suited to SNES MODE7-style graphics routines. For example, on each core, INTERP0 can provide a stream of tile lookups for some affine transform, and INTERP1 can provide offsets into the tiles for the same transform.

2.3.1.6.1. Lane Operations

图8. 一个插值器。两个累加器寄存器及三个基址寄存器均支持处理器单周期读写访问。插值器由两个通道组成，分别对两个累加器执行掩码、移位及符号扩展操作。通过将中间移位/掩码值加至三个基址寄存器，生成三种可能结果。从左至右，每个通道的多路复用器由CTR寄存器中的以下标志位控制：



处理器能在一个周期内读写任意插值器寄存器，结果于下一周期可用。处理器亦可通过写入对应`ACCUMx_ADD`寄存器，对两个累加器`ACCUM0`或`ACCUM1`之一执行加法运算。

`CROSS_RESULT`、
`CROSS_INPUT`、
`SIGNED`、`ADD_RAW`

这三种结果在只读寄存器`PEEK0`、`PEEK1`和`PEEK2`中可用。从上述寄存器读取不会改变插值器的状态。结果同样在寄存器`POP0`、`POP1`、`POP2`处通过别名可访问；从`POPX`别名读取返回的结果与相应的`PEEKX`相同，同时将该通道结果写回累加器。每次读取结果时，可用此方法推进插值器的状态。

此外，插值器支持两个值之间的简单分数混合，以及将值限制在指定范围内。

以下示例展示了弹出一个通道结果以实现简单迭代反馈的基础示例。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第11至23行

```

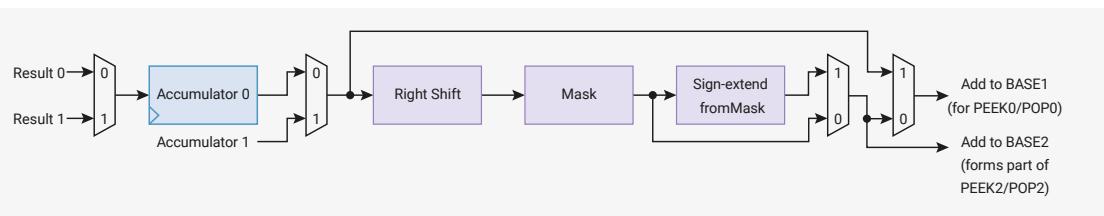
11 void times_table() {
12     puts("9的乘法表：");
13
14     // 在此核心初始化 interp0 通道 0
15     interp_config cfg = interp_default_config();
16     interp_set_config(interp0, 0, &cfg);
17
18     interp0->accum[0] = 0;
19     interp0->base[0] = 9;
20
21     for (int i = 0; i < 10; ++i)
22         printf("%d\n", interp0->pop[0]);
23 }
```

注意

纯属巧合的是，插值器极其适合用于模拟SNES MODE7风格的图形例程。例如，在每个核心上，INTERP0可提供某个仿射变换的图块查找流，而INTERP1则可为相同变换提供图块偏移。

2.3.1.6.1. 通道操作

Figure 9. Each lane of each interpolator can be configured to perform mask, shift and sign-extension on one of the accumulators. This is fed into adders which produces final results, which may optionally be fed back into the accumulators with each read. The datapath can be configured using a handful of 32-bit multiplexers. From left to right, these are controlled by the following CTRL flags: CROSS_RESULT, CROSS_INPUT, SIGNED, ADD_RAW.



Each lane performs these three operations, in sequence:

- A right shift by `CTRL_LANEx_SHIFT` (0 to 31 bits)
- A mask of bits from `CTRL_LANEx_MASK_LSB` to `CTRL_LANEx_MASK_MSB` inclusive (each ranging from bit 0 to bit 31)
- A sign extension from the top of the mask, i.e. take bit `CTRL_LANEx_MASK_MSB` and OR it into all more-significant bits, if `CTRL_LANEx_SIGNED` is set

For example, if:

- `ACCUM0 = 0xdeadbeef`
- `CTRL_LANE0_SHIFT = 8`
- `CTRL_LANE0_MASK_LSB = 4`
- `CTRL_LANE0_MASK_MSB = 7`
- `CTRL_SIGNED = 1`

Then lane 0 would produce the following results at each stage:

- Right shift by 8 to produce `0x00deadbe`
- Mask bits 7 to 4 to produce `0x00deadbe & 0x000000f0 = 0x000000b0`
- Sign-extend up from bit 7 to produce `0xffffffffb0`

In software:

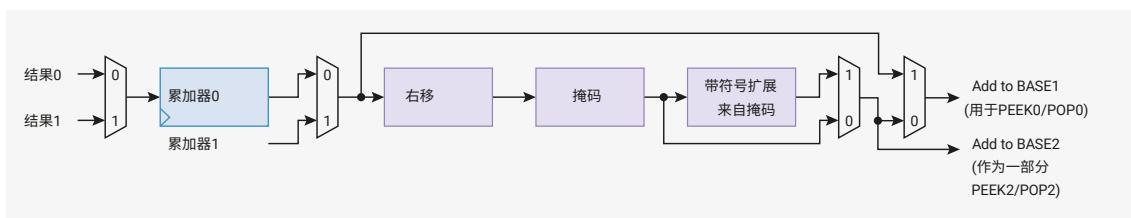
Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 25 - 46

```

25 void moving_mask() {
26     interp_config cfg = interp_default_config();
27     interp0->accum[0] = 0x1234abcd;
28
29     puts("Masking:");
30     printf("ACCUM0 = %08x\n", interp0->accum[0]);
31     for (int i = 0; i < 8; ++i) {
32         // LSB, then MSB. These are inclusive, so 0,31 means "the entire 32 bit register"
33         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
34         interp_set_config(interp0, 0, &cfg);
35         // Reading from ACCUMx_ADD returns the raw lane shift and mask value, without BASEx
36         // added
37         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
38     }
39
39     puts("Masking with sign extension:");
40     interp_config_set_signed(&cfg, true);
41     for (int i = 0; i < 8; ++i) {
42         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
43         interp_set_config(interp0, 0, &cfg);
44         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
45     }
46 }
```

The above example should print:

图9。每个插值器的每个通道均可配置为对指定累加器执行掩码、移位及符号扩展操作。该结果被送入加法器生成最终结果，且该结果可选择性地在每次读取时反馈回累加器。数据通路可通过少量32位多路复用器进行配置。从左至右，这些由以下CTRL标志控制：



每条通路按顺序执行以下三项操作：

- 右移`CTRL_LANEx_SHIFT`位（0至31位）
- 对从`CTRL_LANEx_MASK_LSB`至`CTRL_LANEx_MASK_MSB`（含）范围内的位进行掩码操作（各位均介于0至31位）
- 从掩码最高位进行符号扩展，即若设置了`CTRL_LANEx_SIGNED`，则取位`CTRL_LANEx_MASK_MSB`并将其通过或运算扩展到所有更高有效位

例如，若：

`CROSS_RESULT`、
`CROSS_INPUT`、
`SIGNED`、`ADD_RAW`。

- `ACCUM0 = 0xdeadbeef`
- `CTRL_LANE0_SHIFT = 8`
- `CTRL_LANE0_MASK_LSB = 4`
- `CTRL_LANE0_MASK_MSB = 7`
- `CTRL_SIGNED = 1`

则通路0在每个阶段将产生以下结果：

- 右移8位得到`0x00deadbe`
- 对位7至4加掩码得到`0x00deadbe & 0x000000f0 = 0x000000b0`
- 从位7进行符号扩展得到`0xffffffffb0`

在软件中：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第25至46行

```

25 void moving_mask() {
26     interp_config cfg = interp_default_config();
27     interp0->accum[0] = 0x1234abcd;
28
29     puts("掩码：");
30     printf("ACCUM0 = %08x\n", interp0->accum[0]);
31     for (int i = 0; i < 8; ++i) {
32         // 先最低有效位，再最高有效位。范围包含端点，因此0, 31表示“整个32位寄存器”
33         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
34         interp_set_config(interp0, 0, &cfg);
35         // 从 ACCUMx_ADD 读取返回原始通道移位和掩码值，未加 BASEx
36
37         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
38     }
39
39     puts("带符号扩展的掩码：");
40     interp_config_set_signed(&cfg, true);
41     for (int i = 0; i < 8; ++i) {
42         interp_config_set_mask(&cfg, i * 4, i * 4 + 3);
43         interp_set_config(interp0, 0, &cfg);
44         printf("Nibble %d: %08x\n", i, interp0->add_raw[0]);
45     }
46 }
```

上述示例应打印：

```

ACCUM0 = 1234abcd
Nibble 0: 0000000d
Nibble 1: 000000c0
Nibble 2: 0000b000
Nibble 3: 000a0000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000
Masking with sign extension:
Nibble 0: ffffffd
Nibble 1: ffffffc0
Nibble 2: fffffb00
Nibble 3: fffffa000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000

```

Changing the result and input multiplexers can create feedback between the accumulators. This is useful e.g. for audio dithering.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 48 - 66

```

48 void cross_lanes() {
49     interp_config cfg = interp_default_config();
50     interp_config_set_cross_result(&cfg, true);
51     // ACCUM0 gets lane 1 result:
52     interp_set_config(interp0, 0, &cfg);
53     // ACCUM1 gets lane 0 result:
54     interp_set_config(interp0, 1, &cfg);
55
56     interp0->accum[0] = 123;
57     interp0->accum[1] = 456;
58     interp0->base[0] = 1;
59     interp0->base[1] = 0;
60     puts("Lane result crossover:");
61     for (int i = 0; i < 10; ++i) {
62         uint32_t peek0 = interp0->peek[0];
63         uint32_t pop1 = interp0->pop[1];
64         printf("PEEK0, POP1: %d, %d\n", peek0, pop1);
65     }
66 }

```

This should print:

```

PEEK0, POP1: 124, 456
PEEK0, POP1: 457, 124
PEEK0, POP1: 125, 457
PEEK0, POP1: 458, 125
PEEK0, POP1: 126, 458
PEEK0, POP1: 459, 126
PEEK0, POP1: 127, 459
PEEK0, POP1: 460, 127
PEEK0, POP1: 128, 460
PEEK0, POP1: 461, 128

```

```

ACCUM0 = 1234abcd
Nibble 0: 0000000d
Nibble 1: 000000c0
Nibble 2: 00000b00
Nibble 3: 0000a000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000
带符号扩展的掩码:
Nibble 0: ffffffd
Nibble 1: ffffffc0
Nibble 2: ffffffb00
Nibble 3: fffffa000
Nibble 4: 00040000
Nibble 5: 00300000
Nibble 6: 02000000
Nibble 7: 10000000

```

更改结果和输入多路复用器能够在累加器之间形成反馈，此功能在音频抖动处理中尤为有用。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp.c 第48至66行

```

48 void cross_lanes() {
49     interp_config cfg = interp_default_config();
50     interp_config_set_cross_result(&cfg, true);
51     // ACCUM0 获取通道1的结果:
52     interp_set_config(interp0, 0, &cfg);
53     // ACCUM1 获取通道0的结果:
54     interp_set_config(interp0, 1, &cfg);
55
56     interp0->accum[0] = 123;
57     interp0->accum[1] = 456;
58     interp0->base[0] = 1;
59     interp0->base[1] = 0;
60     puts("通道结果交叉: ");
61     for (int i = 0; i < 10; ++i) {
62         uint32_t peek0 = interp0->peek[0];
63         uint32_t pop1 = interp0->pop[1];
64         printf("PEEK0, POP1: %d, %d\n", peek0, pop1);
65     }
66 }

```

应打印：

```

PEEK0, POP1: 124, 456
PEEK0, POP1: 457, 124
PEEK0, POP1: 125, 457
PEEK0, POP1: 458, 125
PEEK0, POP1: 126, 458
PEEK0, POP1: 459, 126
PEEK0, POP1: 127, 459
PEEK0, POP1: 460, 127
PEEK0, POP1: 128, 460
PEEK0, POP1: 461, 128

```

2.3.1.6.2. Blend Mode

Blend mode is available on `INTERP0` on each core, and is enabled by the `CTRL_LANE0_BLEND` control flag. It performs linear interpolation, which we define as follows:

$$x = x_0 + \alpha(x_1 - x_0), \text{ for } 0 \leq \alpha < 1$$

Where x_0 is the register `BASE0`, x_1 is the register `BASE1`, and α is a fractional value formed from the least significant 8 bits of the lane 1 shift and mask value.

Blend mode has the following differences from normal mode:

- `PEEK0`, `POP0` return the 8-bit alpha value (the 8 LSBs of the lane 1 shift and mask value), with zeroes in result bits 31 down to 24.
- `PEEK1`, `POP1` return the linear interpolation between `BASE0` and `BASE1`
- `PEEK2`, `POP2` do not include lane 1 result in the addition (i.e. it is `BASE0 + lane 0 shift and mask value`)

The result of the linear interpolation is equal to `BASE0` when the alpha value is 0, and equal to `BASE0 + 255/256 * (BASE1 - BASE0)` when the alpha value is all-ones.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 68 - 87

```

68 void simple_blend1() {
69     puts("Simple blend 1:");
70
71     interp_config cfg = interp_default_config();
72     interp_config_set_blend(&cfg, true);
73     interp_set_config(interp0, 0, &cfg);
74
75     cfg = interp_default_config();
76     interp_set_config(interp0, 1, &cfg);
77
78     interp0->base[0] = 500;
79     interp0->base[1] = 1000;
80
81     for (int i = 0; i <= 6; i++) {
82         // set fraction to value between 0 and 255
83         interp0->accum[1] = 255 * i / 6;
84         // ≈ 500 + (1000 - 500) * i / 6;
85         printf("%d\n", (int) interp0->peek[1]);
86     }
87 }
```

This should print (note the 255/256 resulting in 998 not 1000):

```

500
582
666
748
832
914
998
```

`CTRL_LANE1_SIGNED` controls whether `BASE0` and `BASE1` are sign-extended for this interpolation (this sign extension is required because the interpolation produces an intermediate product value 40 bits in size). `CTRL_LANE0_SIGNED` continues to control the sign extension of the lane 0 intermediate result in `PEEK2`, `POP2` as normal.

2.3.1.6.2 混合模式

混合模式可在每个核心的 `INTERP0` 上使用，且通过 `CTRL_LANE0_BLEND` 控制标志启用。其执行线性插值，定义如下：

$$x = x_0 + \alpha(x_1 - x_0), \text{ for } 0 \leq \alpha < 1$$

其中为寄存器 `BASE0`，为寄存器 `BASE1`，且为由通道1的移位和掩码值最低有效8位构成的分数值。

混合模式与普通模式的区别如下：

- `PEEK0` 与 `POP0` 返回8位透明度值（即通道1移位和掩码值的最低8位），结果的第31位至第24位均为零。

- `PEEK1` 与 `POP1` 返回 `BASE0` 和 `BASE1` 之间的线性插值值。
- `PEEK2`, `POP2` 在加法中不包含车道1的结果（即它是 `BASE2` + 车道0的移位和掩码值）

当 `alpha` 值为0时，线性插值的结果等于 `BASE0`；当 `alpha` 值全为1时，结果等于 `BASE0 + 255/256 * (BASE1 - BASE0)`

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第68至87行

```

68 void simple_blend1() {
69     puts("简单混合1: ");
70
71     interp_config cfg = interp_default_config();
72     interp_config_set_blend(&cfg, true);
73     interp_set_config(interp0, 0, &cfg);
74
75     cfg = interp_default_config();
76     interp_set_config(interp0, 1, &cfg);
77
78     interp0->base[0] = 500;
79     interp0->base[1] = 1000;
80
81     for (int i = 0; i <= 6; i++) {
82         // 将fraction设置为介于0至255之间的值
83         interp0->accum[1] = 255 * i / 6;
84         // ≈ 500 + (1000 - 500) * i / 6;
85         printf("%d\n", (int) interp0->peek[1]);
86     }
87 }
```

应当打印（注意 `255/256` 导致结果为 `998`，而非 `1000`）：

```

500
582
666
748
832
914
998
```

`CTRL_LANE1_SIGNED` 控制 `BASE0` 与 `BASE1` 是否进行符号扩展以完成此插值（该符号扩展必须，因为插值生成了一个40位大小的中间乘积值）。`CTRL_LANE0_SIGNED` 继续正常控制 `PEEK2` 与 `POP2` 中 lane 0 中间结果的符号扩展。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 90 - 121

```

90 void print_simple_blend2_results(bool is_signed) {
91     // lane 1 signed flag controls whether base 0/1 are treated as signed or unsigned
92     interp_config cfg = interp_default_config();
93     interp_config_set_signed(&cfg, is_signed);
94     interp_set_config(interp0, 1, &cfg);
95
96     for (int i = 0; i <= 6; i++) {
97         interp0->accum[1] = 255 * i / 6;
98         if (is_signed) {
99             printf("%d\n", (int) interp0->peek[1]);
100        } else {
101            printf("0x%08x\n", (uint) interp0->peek[1]);
102        }
103    }
104 }
105
106 void simple_blend2() {
107     puts("Simple blend 2:");
108
109     interp_config cfg = interp_default_config();
110     interp_config_set_blend(&cfg, true);
111     interp_set_config(interp0, 0, &cfg);
112
113     interp0->base[0] = (uint32_t) -1000;
114     interp0->base[1] = 1000;
115
116     puts("signed:");
117     print_simple_blend2_results(true);
118
119     puts("unsigned:");
120     print_simple_blend2_results(false);
121 }
```

This should print:

```

signed:
-1000
-672
-336
-8
328
656
992
unsigned:
0xfffffc18
0xd5ffffd60
0xaafffeb0
0x80fffff8
0x56000148
0x2c000290
0x010003e0
```

Finally, in blend mode when using the **BASE_1AND0** register to send a 16-bit value to each of **BASE0** and **BASE1** with a single 32-bit write, the sign-extension of these 16-bit values to full 32-bit values during the write is controlled by **CTRL_LANE1_SIGNED** for both bases, as opposed to non-blend-mode operation, where **CTRL_LANE0_SIGNED** affects extension into **BASE0** and **CTRL_LANE1_SIGNED** affects extension into **BASE1**.

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第90至121行

```

90 void print_simple_blend2_results(bool is_signed) {
91     //通道1的有符号标志控制基数0/1是否被视为有符号或无符号
92     interp_config cfg = interp_default_config();
93     interp_config_set_signed(&cfg, is_signed);
94     interp_set_config(interp0, 1, &cfg);
95
96     for (int i = 0; i <= 6; i++) {
97         interp0->accum[1] = 255 * i / 6;
98         if (is_signed) {
99             printf("%d\n", (int) interp0->peek[1]);
100        } else {
101            printf("0x%08x\n", (uint) interp0->peek[1]);
102        }
103    }
104 }
105
106 void simple_blend2() {
107     puts("简单混合2:");
108
109     interp_config cfg = interp_default_config();
110     interp_config_set_blend(&cfg, true);
111     interp_set_config(interp0, 0, &cfg);
112
113     interp0->base[0] = (uint32_t) -1000;
114     interp0->base[1] = 1000;
115
116     puts("有符号:");
117     print_simple_blend2_results(true);
118
119     puts("unsigned:");
120     print_simple_blend2_results(false);
121 }
```

应打印：

```

signed:
-1000
-672
-336
-8
328
656
992
unsigned:
0xfffffc18
0xd5ffffd60
0xaafffeb0
0x80fffff8
0x56000148
0x2c000290
0x010003e0
```

最后，在混合模式下，使用 **BASE_1AND0** 寄存器通过单次32位写入向 **BASE0** 和 **BASE1** 各发送16位值时，这些16位值向完整32位值的符号扩展由**CTRL_LANE1_SIGNED**控制，适用于两个基寄存器。相比之下，非混合模式操作中，**CTRL_LANE0_SIGNED**控制向**BASE0**的扩展，**CTRL_LANE1_SIGNED**控制向**BASE1**的扩展。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 124 - 145

```

124 void simple_blend3() {
125     puts("Simple blend 3:");
126
127     interp_config cfg = interp_default_config();
128     interp_config_set_blend(&cfg, true);
129     interp_set_config(interp0, 0, &cfg);
130
131     cfg = interp_default_config();
132     interp_set_config(interp0, 1, &cfg);
133
134     interp0->accum[1] = 128;
135     interp0->base01 = 0x30005000;
136     printf("0x%08x\n", (int) interp0->peek[1]);
137     interp0->base01 = 0xe000f000;
138     printf("0x%08x\n", (int) interp0->peek[1]);
139
140     interp_config_set_signed(&cfg, true);
141     interp_set_config(interp0, 1, &cfg);
142
143     interp0->base01 = 0xe000f000;
144     printf("0x%08x\n", (int) interp0->peek[1]);
145 }
```

This should print:

```

0x00004000
0x0000e800
0xffffe800
```

2.3.1.6.3. Clamp Mode

Clamp mode is available on **INTERP1** on each core, and is enabled by the **CTRL_LANE0_CLAMP** control flag. In clamp mode, the **PEEK0/POP0** result is the lane value (shifted, masked, sign-extended **ACCUM0**) clamped between **BASE0** and **BASE1**. In other words, if the lane value is greater than **BASE1**, a value of **BASE1** is produced; if less than **BASE0**, a value of **BASE0** is produced; otherwise, the value passes through. No addition is performed. The signedness of these comparisons is controlled by the **CTRL_LANE0_SIGNED** flag.

Other than this, the interpolator behaves the same as in normal mode.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 193 - 211

```

193 void clamp() {
194     puts("Clamp:");
195     interp_config cfg = interp_default_config();
196     interp_config_set_clamp(&cfg, true);
197     interp_config_set_shift(&cfg, 2);
198     // set mask according to new position of sign bit..
199     interp_config_set_mask(&cfg, 0, 29);
200     // ...so that the shifted value is correctly sign extended
201     interp_config_set_signed(&cfg, true);
202     interp_set_config(interp1, 0, &cfg);
203
204     interp1->base[0] = 0;
205     interp1->base[1] = 255;
206
207     for (int i = -1024; i <= 1024; i += 256) {
```

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第124至145行

```

124 void simple_blend3() {
125     puts("简单混合3:");
126
127     interp_config cfg = interp_default_config();
128     interp_config_set_blend(&cfg, true);
129     interp_set_config(interp0, 0, &cfg);
130
131     cfg = interp_default_config();
132     interp_set_config(interp0, 1, &cfg);
133
134     interp0->accum[1] = 128;
135     interp0->base01 = 0x30005000;
136     printf("0x%08x\n", (int) interp0->peek[1]);
137     interp0->base01 = 0xe000f000;
138     printf("0x%08x\n", (int) interp0->peek[1]);
139
140     interp_config_set_signed(&cfg, true);
141     interp_set_config(interp0, 1, &cfg);
142
143     interp0->base01 = 0xe000f000;
144     printf("0x%08x\n", (int) interp0->peek[1]);
145 }
```

应打印：

```

0x00004000
0x0000e800
0xfffffe800
```

2.3.1.6.3. 夹紧模式

夹紧模式适用于每个核心上的 `INTERP1`，且由 `CTRL_LANE0_CLAMP` 控制标志启用。在夹紧模式下，`PEEK0/POP0` 的结果为通道值（经过 `ACCU0` 的移位、掩码及符号扩展），该值被限制在 `BASE0` 与 `BASE1` 之间。换言之，若通道值大于 `BASE1`，则结果为 `BASE1`；若小于 `BASE0`，则结果为 `BASE0`；否则，值保持不变。不进行加法操作。上述比较的符号属性由 `CTRL_LANE0_SIGNED` 标志控制。

除此之外，插值器的行为与正常模式一致。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第193至211行

```

193 void clamp() {
194     puts("Clamp:");
195     interp_config cfg = interp_default_config();
196     interp_config_set_clamp(&cfg, true);
197     interp_config_set_shift(&cfg, 2);
198     // 根据符号位的新位置设置掩码。
199     interp_config_set_mask(&cfg, 0, 29);
200     // 以确保移位后的值正确进行符号扩展。
201     interp_config_set_signed(&cfg, true);
202     interp_set_config(interp1, 0, &cfg);
203
204     interp1->base[0] = 0;
205     interp1->base[1] = 255;
206
207     for (int i = -1024; i <= 1024; i += 256) {
```

```

208         interp1->accum[0] = i;
209         printf("%d\t%d\n", i, (int) interp1->peek[0]);
210     }
211 }
```

This should print:

```

-1024    0
-768     0
-512     0
-256     0
0        0
256     64
512    128
768    192
1024   255
```

2.3.1.6.4. Sample Use Case: Linear Interpolation

Linear interpolation is a more complete example of using blend mode in conjunction with other interpolator functionality:

In this example, `ACCUM0` is used to track a fixed point (integer/fraction) position within a list of values to be interpolated. Lane 0 is used to produce an address into the value array for the integer part of the position. The fractional part of the position is shifted to produce a value from 0-255 for the blend. The blend is performed between two consecutive values in the array.

Finally the fractional position is updated via a single write to `ACCUM0_ADD_RAW`.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 147 - 191

```

147 void linear_interpolation() {
148     puts("Linear interpolation:");
149     const int uv_fractional_bits = 12;
150
151     // for lane 0
152     // shift and mask XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (accum 0)
153     // to          0000 0000 000X XXXX XXXX XXXX XXXX XXX0
154     // i.e. non fractional part times 2 (for uint16_t)
155     interp_config cfg = interp_default_config();
156     interp_config_set_shift(&cfg, uv_fractional_bits - 1);
157     interp_config_set_mask(&cfg, 1, 32 - uv_fractional_bits);
158     interp_config_set_blend(&cfg, true);
159     interp_set_config(interp0, 0, &cfg);
160
161     // for lane 1
162     // shift XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (accum 0 via cross input)
163     // to      0000 XXXX XXXX XXXX XXXX FFFF FFFF FFFF
164
165     cfg = interp_default_config();
166     interp_config_set_shift(&cfg, uv_fractional_bits - 8);
167     interp_config_set_signed(&cfg, true);
168     interp_config_set_cross_input(&cfg, true); // signed blending
169     interp_set_config(interp0, 1, &cfg);
170
171     int16_t samples[] = {0, 10, -20, -1000, 500};
172
173     // step is 1/4 in our fractional representation
```

```

208         interp1->accum[0] = i;
209         printf("%d\t%d\n", i, (int) interp1->peek[0]);
210     }
211 }
```

应打印：

```

-1024 0
-768 0
-512 0
-256 0
0 0
256 64
512 128
768 192
1024 255
```

2.3.1.6.4. 示例用例：线性插值

线性插值是结合混合模式及其他插值器功能的更完整示例：

在本示例中，`ACCUM0`用于跟踪待插值数值列表中的定点（整数/小数）位置。

通道0用于生成位置整数部分对应的值数组地址。位置的小数部分被移位以生成0至255范围内的混合值。混合操作在数组中相邻的两个值之间执行。

最后，通过对`ACCUM0_ADD_RAW`的单次写入操作更新小数位置。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第147至191行

```

147 void linear_interpolation() {
148     puts("线性插值:");
149     const int uv_fractional_bits = 12;
150
151     // 通道0
152     // 对XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (累加器0) 进行移位和掩码
153     // 至          0000 0000 000X XXXX XXXX XXXX XXXX XXX0 154
154     // 即整数部分乘以2 (用于uint16_t)
155     interp_config cfg = interp_default_config();
156     interp_config_set_shift(&cfg, uv_fractional_bits - 1);
157     interp_config_set_mask(&cfg, 1, 32 - uv_fractional_bits);
158     interp_config_set_blend(&cfg, true);
159     interp_set_config(interp0, 0, &cfg);
160
161     // 对通道1
162     // 将 XXXX XXXX XXXX XXXX XXXX FFFF FFFF FFFF (通过交叉输入累加为0)
163     // 移位至 0000 XXXX XXXX XXXX XXXX FFFF FFFF FFFF
164
165     cfg = interp_default_config();
166     interp_config_set_shift(&cfg, uv_fractional_bits - 8);
167     interp_config_set_signed(&cfg, true);
168     interp_config_set_cross_input(&cfg, true); // 有符号混合
169     interp_set_config(interp0, 1, &cfg);
170
171     int16_t samples[] = {0, 10, -20, -1000, 500};
172
173     // step 在我们的分数表示中为四分之一
```

```

174     uint step = (1 << uv_fractional_bits) / 4;
175
176     interp0->accum[0] = 0; // initial sample_offset;
177     interp0->base[2] = (uintptr_t) samples;
178     for (int i = 0; i < 16; i++) {
179         // result2 = samples + (lane0 raw result)
180         // i.e. ptr to the first of two samples to blend between
181         int16_t *sample_pair = (int16_t *) interp0->peek[2];
182         interp0->base[0] = sample_pair[0];
183         interp0->base[1] = sample_pair[1];
184         uint32_t peek1 = interp0->peek[1];
185         uint32_t add_raw1 = interp0->add_raw[1];
186         printf("%d\t(%d% between %d and %d)\n", (int) peek1,
187                100 * (add_raw1 & 0xff) / 0xff,
188                sample_pair[0], sample_pair[1]);
189         interp0->add_raw[0] = step;
190     }
191 }
```

This should print:

```

0      (0% between 0 and 10)
2      (25% between 0 and 10)
5      (50% between 0 and 10)
7      (75% between 0 and 10)
10     (0% between 10 and -20)
2      (25% between 10 and -20)
-5     (50% between 10 and -20)
-13    (75% between 10 and -20)
-20    (0% between -20 and -1000)
-265   (25% between -20 and -1000)
-510   (50% between -20 and -1000)
-755   (75% between -20 and -1000)
-1000  (0% between -1000 and 500)
-625   (25% between -1000 and 500)
-250   (50% between -1000 and 500)
125    (75% between -1000 and 500)
```

This method is used for fast approximate audio upscaling in the SDK

2.3.1.6.5. Sample Use Case: Simple Affine Texture Mapping

Simple affine texture mapping can be implemented by using fixed point arithmetic for texture coordinates, and stepping a fixed amount in each coordinate for every pixel in a scanline. The integer part of the texture coordinates are used to form an address within the texture to lookup a pixel colour.

By using two lanes, all three base values and the `CTRL_LANEx_ADD_RAW` flag, it is possible to reduce what would be quite an expensive CPU operation to a single cycle iteration using the interpolator.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c Lines 214 - 272

```

214 void texture_mapping_setup(uint8_t *texture, uint texture_width_bits, uint
215                             texture_height_bits,
216                             uint uv_fractional_bits) {
217     interp_config cfg = interp_default_config();
218     // set add_raw flag to use raw (un-shifted and un-masked) lane accumulator value when
219     // adding
220     // it to the lane base to make the lane result
```

```

174     uint step = (1 << uv_fractional_bits) / 4;
175
176     interp0->accum[0] = 0; // 初始 sample_offset;
177     interp0->base[2] = (uintptr_t) samples;
178     for (int i = 0; i < 16; i++) {
179         // result2 = samples + (lane0 原始结果)
180         // 即指向两个需混合样本中第一个样本的指针
181         int16_t *sample_pair = (int16_t *) interp0->peek[2];
182         interp0->base[0] = sample_pair[0];
183         interp0->base[1] = sample_pair[1];
184         uint32_t peek1 = interp0->peek[1];
185         uint32_t add_raw1 = interp0->add_raw[1];
186         printf("%d\t(%d%% 介于 %d 和 %d 之间)\n", (int) peek1,
187                100 * (add_raw1 & 0xff) / 0xff,
188                sample_pair[0], sample_pair[1]);
189         interp0->add_raw[0] = step;
190     }
191 }
```

应打印：

```

0      (0% 介于 0 和 10 之间)
2      (25% 介于 0 和 10 之间)
5      (50% 介于 0 和 10 之间)
7      (75% 介于 0 和 10 之间)
10     (0% 介于 10 和 -20 之间)
2      (25% 介于 10 和 -20 之间)
-5     (50% 介于 10 和 -20 之间)
-13    (75% 介于 10 和 -20 之间)
-20    (0% 介于 -20 和 -1000 之间)
-265   (25% 介于 -20 和 -1000 之间)
-510   (-20 与 -1000 之间的 50%)
-755   (-20 与 -1000 之间的 75%)
-1000  (-1000 与 500 之间的 0%)
-625   (-1000 与 500 之间的 25%)
-250   (-1000 与 500 之间的 50%)
125    (-1000 与 500 之间的 75%)
```

此方法用于 SDK 中的快速近似音频上采样。

2.3.1.6.5. 示例用例：简单仿射纹理映射

简单仿射纹理映射可通过对纹理坐标使用定点运算来实现，并在扫描线的每个像素处按固定步长更新坐标。纹理坐标的整数部分用于在纹理中形成地址，进而查找像素颜色。

通过使用两条通道、所有三个基值及`CTRL_LANEx_ADD_RAW`标志，可将原本较为昂贵的 CPU 操作简化为利用插值器执行的单周期迭代。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/interp/hello_interp/hello_interp.c 第 214 至 272 行

```

214 void texture_mapping_setup(uint8_t *texture, uint texture_width_bits, uint
215                             texture_height_bits,
216                             uint uv_fractional_bits) {
217     interp_config cfg = interp_default_config();
218     // 设置 add_raw 标志以在加法时使用原始（未移位且未屏蔽）的通道累加器值
219
220     // 将其加至通道基值以生成通道结果
```

```

219     interp_config_set_add_raw(&cfg, true);
220     interp_config_set_shift(&cfg, uv_fractional_bits);
221     interp_config_set_mask(&cfg, 0, texture_width_bits - 1);
222     interp_set_config(interp0, 0, &cfg);
223
224     interp_config_set_shift(&cfg, uv_fractional_bits - texture_width_bits);
225     interp_config_set_mask(&cfg, texture_width_bits, texture_width_bits +
        texture_height_bits - 1);
226     interp_set_config(interp0, 1, &cfg);
227
228     interp0->base[2] = (uintptr_t) texture;
229 }
230
231 void texture_mapped_span(uint8_t *output, uint32_t u, uint32_t v, uint32_t du, uint32_t dv,
   uint count) {
232     // u, v are texture coordinates in fixed point with uv_fractional_bits fractional bits
233     // du, dv are texture coordinate steps across the span in same fixed point.
234     interp0->accum[0] = u;
235     interp0->base[0] = du;
236     interp0->accum[1] = v;
237     interp0->base[1] = dv;
238     for (uint i = 0; i < count; i++) {
239         // equivalent to
240         // uint32_t sm_result0 = (accum0 >> uv_fractional_bits) & (1 << (texture_width_bits -
        1));
241         // uint32_t sm_result1 = (accum1 >> uv_fractional_bits) & (1 << (texture_height_bits -
        1));
242         // uint8_t *address = texture + sm_result0 + (sm_result1 << texture_width_bits);
243         // output[i] = *address;
244         // accum0 = du + accum0;
245         // accum1 = dv + accum1;
246
247         // result2 is the texture address for the current pixel;
248         // popping the result advances to the next iteration
249         output[i] = *(uint8_t *) interp0->pop[2];
250     }
251 }
252
253 void texture_mapping() {
254     puts("Affine Texture mapping (with texture wrap):");
255
256     uint8_t texture[] = {
257         0x00, 0x01, 0x02, 0x03,
258         0x10, 0x11, 0x12, 0x13,
259         0x20, 0x21, 0x22, 0x23,
260         0x30, 0x31, 0x32, 0x33,
261     };
262     // 4x4 texture
263     texture_mapping_setup(texture, 2, 2, 16);
264     uint8_t output[12];
265     uint32_t du = 65536 / 2; // step of 1/2
266     uint32_t dv = 65536 / 3; // step of 1/3
267     texture_mapped_span(output, 0, 0, du, dv, 12);
268
269     for (uint i = 0; i < 12; i++) {
270         printf("0x%02x\n", output[i]);
271     }
272 }

```

This should print:

```

219     interp_config_set_add_raw(&cfg, true);
220     interp_config_set_shift(&cfg, uv_fractional_bits);
221     interp_config_set_mask(&cfg, 0, texture_width_bits - 1);
222     interp_set_config(interp0, 0, &cfg);
223
224     interp_config_set_shift(&cfg, uv_fractional_bits - texture_width_bits);
225     interp_config_set_mask(&cfg, texture_width_bits, texture_width_bits +
        texture_height_bits - 1);
226     interp_set_config(interp0, 1, &cfg);
227
228     interp0->base[2] = (uintptr_t) texture;
229 }
230
231 void texture_mapped_span(uint8_t *output, uint32_t u, uint32_t v, uint32_t du, uint32_t dv,
   uint count) {
232     // u、v为具有uv_fractional_bits小数位的固定点纹理坐标
233     // du、dv为跨越该片段的纹理坐标步长，采用相同的固定点表示法。
234     interp0->accum[0] = u;
235     interp0->base[0] = du;
236     interp0->accum[1] = v;
237     interp0->base[1] = dv;
238     for (uint i = 0; i < count; i++) {
239         // 等同于
240         // uint32_t sm_result0 = (accum0 >> uv_fractional_bits) & (1 << (texture_width_bits -
        1));
241         // uint32_t sm_result1 = (accum1 >> uv_fractional_bits) & (1 << (texture_height_bits -
        1));
242         // uint8_t *address = texture + sm_result0 + (sm_result1 << texture_width_bits);
243         // output[i] = *address;
244         // accum0 = du + accum0;
245         // accum1 = dv + accum1;
246
247         // result2是当前像素的纹理地址;
248         // 弹出结果以便进行下一次迭代
249         output[i] = *(uint8_t *) interp0->pop[2];
250     }
251 }
252
253 void texture_mapping() {
254     puts("仿射纹理映射（带纹理重复）：");
255
256     uint8_t texture[] = {
257         0x00, 0x01, 0x02, 0x03,
258         0x10, 0x11, 0x12, 0x13,
259         0x20, 0x21, 0x22, 0x23,
260         0x30, 0x31, 0x32, 0x33,
261     };
262     // 4x4纹理
263     texture_mapping_setup(texture, 2, 2, 16);
264     uint8_t output[12];
265     uint32_t du = 65536 / 2; // 步长为1/2
266     uint32_t dv = 65536 / 3; // 步长为1/3
267     texture_mapped_span(output, 0, 0, du, dv, 12);
268
269     for (uint i = 0; i < 12; i++) {
270         printf("0x%02x\n", output[i]);
271     }
272 }

```

应打印：

```

0x00
0x00
0x01
0x01
0x12
0x12
0x12
0x13
0x23
0x20
0x20
0x31
0x31

```

2.3.1.7. List of Registers

The SIO registers start at a base address of `0xd0000000` (defined as `SIO_BASE` in SDK).

Table 16. List of SIO registers

Offset	Name	Info
0x000	CPUID	Processor core identifier
0x004	GPIO_IN	Input value for GPIO pins
0x008	GPIO_HI_IN	Input value for QSPI pins
0x010	GPIO_OUT	GPIO output value
0x014	GPIO_OUT_SET	GPIO output value set
0x018	GPIO_OUT_CLR	GPIO output value clear
0x01c	GPIO_OUT_XOR	GPIO output value XOR
0x020	GPIO_OE	GPIO output enable
0x024	GPIO_OE_SET	GPIO output enable set
0x028	GPIO_OE_CLR	GPIO output enable clear
0x02c	GPIO_OE_XOR	GPIO output enable XOR
0x030	GPIO_HI_OUT	QSPI output value
0x034	GPIO_HI_OUT_SET	QSPI output value set
0x038	GPIO_HI_OUT_CLR	QSPI output value clear
0x03c	GPIO_HI_OUT_XOR	QSPI output value XOR
0x040	GPIO_HI_OE	QSPI output enable
0x044	GPIO_HI_OE_SET	QSPI output enable set
0x048	GPIO_HI_OE_CLR	QSPI output enable clear
0x04c	GPIO_HI_OE_XOR	QSPI output enable XOR
0x050	FIFO_ST	Status register for inter-core FIFOs (mailboxes).
0x054	FIFO_WR	Write access to this core's TX FIFO
0x058	FIFO_RD	Read access to this core's RX FIFO
0x05c	SPINLOCK_ST	Spinlock state
0x060	DIV_UDIVIDEND	Divider unsigned dividend

```

0x00
0x00
0x01
0x01
0x12
0x12
0x12
0x13
0x23
0x20
0x20
0x20
0x31
0x31

```

2.3.1.7. 寄存器列表

SIO 寄存器地址起始于 **0xd0000000** (在 SDK 中定义为 SIO_BASE)。

表16. SIO
寄存器列表

偏移量	名称	说明
0x000	CPUID	处理器核心标识符
0x004	GPIO_IN	GPIO引脚输入值
0x008	GPIO_HI_IN	QSPI引脚输入值
0x010	GPIO_OUT	GPIO输出值
0x014	GPIO_OUT_SET	GPIO输出值设置
0x018	GPIO_OUT_CLR	GPIO输出值清除
0x01c	GPIO_OUT_XOR	GPIO输出值异或
0x020	GPIO_OE	GPIO输出使能
0x024	GPIO_OE_SET	GPIO输出使能设置
0x028	GPIO_OE_CLR	GPIO输出使能清除
0x02c	GPIO_OE_XOR	GPIO 输出使能 XOR
0x030	GPIO_HI_OUT	QSPI 输出值
0x034	GPIO_HI_OUT_SET	QSPI 输出值设置
0x038	GPIO_HI_OUT_CLR	QSPI 输出值清除
0x03c	GPIO_HI_OUT_XOR	QSPI 输出值 XOR
0x040	GPIO_HI_OE	QSPI 输出使能
0x044	GPIO_HI_OE_SET	QSPI 输出使能设置
0x048	GPIO_HI_OE_CLR	QSPI 输出使能清除
0x04c	GPIO_HI_OE_XOR	QSPI 输出使能 XOR
0x050	FIFO_ST	核间FIFO（邮箱）状态寄存器。
0x054	FIFO_WR	该核TX FIFO的写访问权限
0x058	FIFO_RD	该核RX FIFO的读访问权限
0x05c	SPINLOCK_ST	自旋锁状态
0x060	DIV_UDIVIDEND	除法器无符号被除数

Offset	Name	Info
0x064	DIV_UDIVISOR	Divider unsigned divisor
0x068	DIV_SDIVIDEND	Divider signed dividend
0x06c	DIV_SDIVISOR	Divider signed divisor
0x070	DIV_QUOTIENT	Divider result quotient
0x074	DIV_REMAINDER	Divider result remainder
0x078	DIV_CSR	Control and status register for divider.
0x080	INTERP0_ACCUM0	Read/write access to accumulator 0
0x084	INTERP0_ACCUM1	Read/write access to accumulator 1
0x088	INTERP0_BASE0	Read/write access to BASE0 register.
0x08c	INTERP0_BASE1	Read/write access to BASE1 register.
0x090	INTERP0_BASE2	Read/write access to BASE2 register.
0x094	INTERP0_POP_LANE0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).
0x098	INTERP0_POP_LANE1	Read LANE1 result, and simultaneously write lane results to both accumulators (POP).
0x09c	INTERP0_POP_FULL	Read FULL result, and simultaneously write lane results to both accumulators (POP).
0x0a0	INTERP0_PEEK_LANE0	Read LANE0 result, without altering any internal state (PEEK).
0x0a4	INTERP0_PEEK_LANE1	Read LANE1 result, without altering any internal state (PEEK).
0x0a8	INTERP0_PEEK_FULL	Read FULL result, without altering any internal state (PEEK).
0x0ac	INTERP0_CTRL_LANE0	Control register for lane 0
0x0b0	INTERP0_CTRL_LANE1	Control register for lane 1
0x0b4	INTERP0_ACCUM0_ADD	Values written here are atomically added to ACCUM0
0x0b8	INTERP0_ACCUM1_ADD	Values written here are atomically added to ACCUM1
0x0bc	INTERP0_BASE_1AND0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously.
0x0c0	INTERP1_ACCUM0	Read/write access to accumulator 0
0x0c4	INTERP1_ACCUM1	Read/write access to accumulator 1
0x0c8	INTERP1_BASE0	Read/write access to BASE0 register.
0x0cc	INTERP1_BASE1	Read/write access to BASE1 register.
0x0d0	INTERP1_BASE2	Read/write access to BASE2 register.
0x0d4	INTERP1_POP_LANE0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).
0x0d8	INTERP1_POP_LANE1	Read LANE1 result, and simultaneously write lane results to both accumulators (POP).
0x0dc	INTERP1_POP_FULL	Read FULL result, and simultaneously write lane results to both accumulators (POP).
0x0e0	INTERP1_PEEK_LANE0	Read LANE0 result, without altering any internal state (PEEK).

偏移量	名称	说明
0x064	DIV_UDIVISOR	除法器无符号除数
0x068	DIV_SDIVIDEND	除法器有符号被除数
0x06c	DIV_SDIVISOR	除法器有符号除数
0x070	DIV_QUOTIENT	除法器结果商
0x074	DIV_REMAINDER	除法器结果余数
0x078	DIV_CSR	除法器控制与状态寄存器。
0x080	INTERP0_ACCUM0	累加器0的读写访问
0x084	INTERP0_ACCUM1	累加器1的读写访问
0x088	INTERP0_BASE0	BASE0寄存器的读写访问。
0x08c	INTERP0_BASE1	BASE1寄存器的读写访问。
0x090	INTERP0_BASE2	BASE2寄存器的读写访问。
0x094	INTERP0_POP_LANE0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作）。
0x098	INTERP0_POP_LANE1	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作）。
0x09c	INTERP0_POP_FULL	读取完整结果，同时将通道结果写入两个累加器（POP）。
0x0a0	INTERP0_PEEK_LANE0	读取LANE0结果，不改变任何内部状态（PEEK）。
0x0a4	INTERP0_PEEK_LANE1	读取LANE1结果，不改变任何内部状态（PEEK）。
0x0a8	INTERP0_PEEK_FULL	读取完整结果，不改变任何内部状态（PEEK）。
0x0ac	INTERP0_CTRL_LANE0	LANE 0 控制寄存器
0x0b0	INTERP0_CTRL_LANE1	LANE 1 控制寄存器
0x0b4	INTERP0_ACCUM0_ADD	写入的值会以原子方式累加至ACCUM0
0x0b8	INTERP0_ACCUM1_ADD	写入的值会以原子方式累加至ACCUM1
0x0bc	INTERP0_BASE_1AND0	写入时，低16位同时写入BASE0，高位写入BASE1。
0x0c0	INTERP1_ACCUM0	累加器0的读写访问
0x0c4	INTERP1_ACCUM1	累加器1的读写访问
0x0c8	INTERP1_BASE0	BASE0寄存器的读写访问。
0x0cc	INTERP1_BASE1	BASE1寄存器的读写访问。
0x0d0	INTERP1_BASE2	BASE2寄存器的读写访问。
0x0d4	INTERP1_POP_LANE0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作）。
0x0d8	INTERP1_POP_LANE1	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作）。
0x0dc	INTERP1_POP_FULL	读取完整结果，同时将通道结果写入两个累加器（POP）。
0x0e0	INTERP1_PEEK_LANE0	读取LANE0结果，不改变任何内部状态（PEEK）。

Offset	Name	Info
0x0e4	INTERP1_PEEK_LANE1	Read LANE1 result, without altering any internal state (PEEK).
0x0e8	INTERP1_PEEK_FULL	Read FULL result, without altering any internal state (PEEK).
0x0ec	INTERP1_CTRL_LANE0	Control register for lane 0
0x0f0	INTERP1_CTRL_LANE1	Control register for lane 1
0x0f4	INTERP1_ACCUM0_ADD	Values written here are atomically added to ACCUM0
0x0f8	INTERP1_ACCUM1_ADD	Values written here are atomically added to ACCUM1
0x0fc	INTERP1_BASE_1AND0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously.
0x100	SPINLOCK0	Spinlock register 0
0x104	SPINLOCK1	Spinlock register 1
0x108	SPINLOCK2	Spinlock register 2
0x10c	SPINLOCK3	Spinlock register 3
0x110	SPINLOCK4	Spinlock register 4
0x114	SPINLOCK5	Spinlock register 5
0x118	SPINLOCK6	Spinlock register 6
0x11c	SPINLOCK7	Spinlock register 7
0x120	SPINLOCK8	Spinlock register 8
0x124	SPINLOCK9	Spinlock register 9
0x128	SPINLOCK10	Spinlock register 10
0x12c	SPINLOCK11	Spinlock register 11
0x130	SPINLOCK12	Spinlock register 12
0x134	SPINLOCK13	Spinlock register 13
0x138	SPINLOCK14	Spinlock register 14
0x13c	SPINLOCK15	Spinlock register 15
0x140	SPINLOCK16	Spinlock register 16
0x144	SPINLOCK17	Spinlock register 17
0x148	SPINLOCK18	Spinlock register 18
0x14c	SPINLOCK19	Spinlock register 19
0x150	SPINLOCK20	Spinlock register 20
0x154	SPINLOCK21	Spinlock register 21
0x158	SPINLOCK22	Spinlock register 22
0x15c	SPINLOCK23	Spinlock register 23
0x160	SPINLOCK24	Spinlock register 24
0x164	SPINLOCK25	Spinlock register 25
0x168	SPINLOCK26	Spinlock register 26
0x16c	SPINLOCK27	Spinlock register 27

偏移量	名称	说明
0x0e4	INTERP1_PEEK_LANE1	读取LANE1结果，不改变任何内部状态（PEEK）。
0x0e8	INTERP1_PEEK_FULL	读取完整结果，不改变任何内部状态（PEEK）。
0x0ec	INTERP1_CTRL_LANE0	LANE 0 控制寄存器
0x0f0	INTERP1_CTRL_LANE1	LANE 1 控制寄存器
0x0f4	INTERP1_ACCUM0_ADD	写入的值会以原子方式累加至ACCUM0
0x0f8	INTERP1_ACCUM1_ADD	写入的值会以原子方式累加至ACCUM1
0x0fc	INTERP1_BASE_1AND0	写入时，低16位同时写入BASE0，高位写入BASE1。
0x100	SPINLOCK0	自旋锁寄存器 0
0x104	SPINLOCK1	自旋锁寄存器1
0x108	SPINLOCK2	自旋锁寄存器2
0x10c	SPINLOCK3	自旋锁寄存器3
0x110	SPINLOCK4	自旋锁寄存器4
0x114	SPINLOCK5	自旋锁寄存器5
0x118	SPINLOCK6	自旋锁寄存器6
0x11c	SPINLOCK7	自旋锁寄存器7
0x120	SPINLOCK8	自旋锁寄存器8
0x124	SPINLOCK9	自旋锁寄存器9
0x128	SPINLOCK10	自旋锁寄存器10
0x12c	SPINLOCK11	自旋锁寄存器11
0x130	SPINLOCK12	自旋锁寄存器12
0x134	SPINLOCK13	自旋锁寄存器13
0x138	SPINLOCK14	自旋锁寄存器14
0x13c	SPINLOCK15	自旋锁寄存器15
0x140	SPINLOCK16	自旋锁寄存器16
0x144	SPINLOCK17	自旋锁寄存器17
0x148	SPINLOCK18	自旋锁寄存器18
0x14c	SPINLOCK19	自旋锁寄存器19
0x150	SPINLOCK20	自旋锁寄存器20
0x154	SPINLOCK21	自旋锁寄存器21
0x158	SPINLOCK22	自旋锁寄存器22
0x15c	SPINLOCK23	自旋锁寄存器23
0x160	SPINLOCK24	自旋锁寄存器24
0x164	SPINLOCK25	自旋锁寄存器 25
0x168	SPINLOCK26	自旋锁寄存器 26
0x16c	SPINLOCK27	自旋锁寄存器 27

Offset	Name	Info
0x170	SPINLOCK28	Spinlock register 28
0x174	SPINLOCK29	Spinlock register 29
0x178	SPINLOCK30	Spinlock register 30
0x17c	SPINLOCK31	Spinlock register 31

SIO: CPUID Register

Offset: 0x000

Description

Processor core identifier

Table 17. CPUID Register

Bits	Description	Type	Reset
31:0	Value is 0 when read from processor core 0, and 1 when read from processor core 1.	RO	-

SIO: GPIO_IN Register

Offset: 0x004

Description

Input value for GPIO pins

Table 18. GPIO_IN Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Input value for GPIO0...29	RO	0x00000000

SIO: GPIO_HI_IN Register

Offset: 0x008

Description

Input value for QSPI pins

Table 19. GPIO_HI_IN Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Input value on QSPI IO in order 0..5: SCLK, SSn, SD0, SD1, SD2, SD3	RO	0x00

SIO: GPIO_OUT Register

Offset: 0x010

Description

GPIO output value

Table 20. GPIO_OUT Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-

偏移量	名称	说明
0x170	SPINLOCK28	自旋锁寄存器 28
0x174	SPINLOCK29	自旋锁寄存器 29
0x178	SPINLOCK30	自旋锁寄存器 30
0x17c	SPINLOCK31	自旋锁寄存器 31

SIO: CPUID 寄存器

偏移: 0x000

描述

处理器核心标识符

表 17. CPUID 寄存器

位	描述	类型	复位值
31:0	从处理器核心 0 读取时, 值为 0; 从处理器核心 1 读取时, 值为 1。	只读	-

SIO: GPIO_IN 寄存器

偏移: 0x004

描述

GPIO引脚输入值

表 18. GPIO_IN 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	GPIO0 至 GPIO29 的输入值	只读	0x00000000

SIO: GPIO_HI_IN 寄存器

偏移: 0x008

描述

QSPI引脚输入值

表 19. GPIO_HI_IN 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	QSPI IO 端口 0 至 5 顺序的输入值: SCLK、SSn、SD0、SD1、SD2、SD3	只读	0x00

SIO: GPIO_OUT 寄存器

偏移: 0x010

描述

GPIO 输出值

表 20. GPIO_OUT 寄存器

位	描述	类型	复位值
31:30	保留。	-	-

Bits	Description	Type	Reset
29:0	Set output level (1/0 → high/low) for GPIO0...29. Reading back gives the last value written, NOT the input value from the pins. If core 0 and core 1 both write to GPIO_OUT simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00000000

SIO: GPIO_OUT_SET Register

Offset: 0x014

Description

GPIO output value set

Table 21.
GPIO_OUT_SET
Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-set on GPIO_OUT, i.e. <code>GPIO_OUT = wdata</code>	WO	0x00000000

SIO: GPIO_OUT_CLR Register

Offset: 0x018

Description

GPIO output value clear

Table 22.
GPIO_OUT_CLR
Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-clear on GPIO_OUT, i.e. <code>GPIO_OUT &= ~wdata</code>	WO	0x00000000

SIO: GPIO_OUT_XOR Register

Offset: 0x01c

Description

GPIO output value XOR

Table 23.
GPIO_OUT_XOR
Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bitwise XOR on GPIO_OUT, i.e. <code>GPIO_OUT ^= wdata</code>	WO	0x00000000

SIO: GPIO_OE Register

Offset: 0x020

Description

GPIO output enable

Table 24. GPIO_OE
Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-

位	描述	类型	复位值
29:0	设置GPIO0至29的输出电平（1/0 → 高/低）。 读回的值为最后写入的值，非引脚上的输入值。 如果核心0和核心1同时写入GPIO_OUT（或写入SET/CLR/XOR别名）， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00000000

SIO: GPIO_OUT_SET寄存器

偏移: 0x014

描述

GPIO输出值设置

表21。
GPIO_OUT_SE
T寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位设置操作，即 <code>GPIO_OUT = wdata</code>	WO	0x00000000

SIO: GPIO_OUT_CLR寄存器

偏移: 0x018

描述

GPIO输出值清除

表22。
GPIO_OUT_CL
R寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位清除操作，即 <code>GPIO_OUT &= ~wdata</code>	WO	0x00000000

SIO: GPIO_OUT_XOR寄存器

偏移: 0x01c

描述

GPIO输出值异或

表 23。
GPIO_OUT_XOR
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对GPIO_OUT执行原子位异或操作，即 <code>GPIO_OUT ^= wdata</code>	WO	0x00000000

SIO: GPIO_OE 寄存器

偏移: 0x020

描述

GPIO 输出使能

表 24. GPIO_OE
寄存器

位	描述	类型	复位值
31:30	保留。	-	-

Bits	Description	Type	Reset
29:0	Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00000000

SIO: GPIO_OE_SET Register

Offset: 0x024

Description

GPIO output enable set

Table 25.
GPIO_OE_SET Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-set on GPIO_OE, i.e. <code>GPIO_OE = wdata</code>	WO	0x00000000

SIO: GPIO_OE_CLR Register

Offset: 0x028

Description

GPIO output enable clear

Table 26.
GPIO_OE_CLR Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-clear on GPIO_OE, i.e. <code>GPIO_OE &= ~wdata</code>	WO	0x00000000

SIO: GPIO_OE_XOR Register

Offset: 0x02c

Description

GPIO output enable XOR

Table 27.
GPIO_OE_XOR Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bitwise XOR on GPIO_OE, i.e. <code>GPIO_OE ^= wdata</code>	WO	0x00000000

SIO: GPIO_HI_OUT Register

Offset: 0x030

Description

QSPI output value

Table 28.
GPIO_HI_OUT Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-

位	描述	类型	复位值
29:0	设置 GPIO0...29 的输出使能（1/0 → 输出/输入）。 读取操作返回最后写入的值。 如果核心0和核心1同时写入 GPIO_OE（或其SET/CLR/XOR别名寄存器）， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00000000

SIO: GPIO_OE_SET 寄存器

偏移: 0x024

描述

GPIO输出使能设置

表 25。
GPIO_OE_SET 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子位置位操作，即 $\text{GPIO_OE} = \text{wdata}$	WO	0x00000000

SIO: GPIO_OE_CLR 寄存器

偏移: 0x028

描述

GPIO输出使能清除

表 26。
GPIO_OE_CLR 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子位清除操作，即 $\text{GPIO_OE} \&= \sim \text{wdata}$	WO	0x00000000

SIO: GPIO_OE_XOR 寄存器

偏移: 0x02c

描述

GPIO 输出使能 XOR

表 27。
GPIO_OE_XOR
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对 GPIO_OE 执行原子按位异或操作，即 $\text{GPIO_OE} \wedge= \text{wdata}$	WO	0x00000000

SIO: GPIO_HI_OUT 寄存器

偏移: 0x030

描述

QSPI 输出值

表 28。
GPIO_HI_OUT 寄存器

位	描述	类型	复位值
31:6	保留。	-	-

Bits	Description	Type	Reset
5:0	Set output level (1/0 → high/low) for QSPI IO0...5. Reading back gives the last value written, NOT the input value from the pins. If core 0 and core 1 both write to GPIO_HI_OUT simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00

SIO: GPIO_HI_OUT_SET Register

Offset: 0x034

Description

QSPI output value set

Table 29.
GPIO_HI_OUT_SET
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bit-set on GPIO_HI_OUT, i.e. <code>GPIO_HI_OUT = wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_CLR Register

Offset: 0x038

Description

QSPI output value clear

Table 30.
GPIO_HI_OUT_CLR
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bit-clear on GPIO_HI_OUT, i.e. <code>GPIO_HI_OUT &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_XOR Register

Offset: 0x03c

Description

QSPI output value XOR

Table 31.
GPIO_HI_OUT_XOR
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bitwise XOR on GPIO_HI_OUT, i.e. <code>GPIO_HI_OUT ^= wdata</code>	WO	0x00

SIO: GPIO_HI_OE Register

Offset: 0x040

Description

QSPI output enable

Table 32. GPIO_HI_OE
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-

位	描述	类型	复位值
5:0	设置 QSPI IO0...5 的输出电平 (1/0 → 高/低)。 读回的值为最后写入的值，非引脚上的输入值。 若核心0与核心1同时写入 GPIO_HI_OUT (或其SET/CLR/XOR别名) ,则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00

SIO: GPIO_HI_OUT_SET 寄存器

偏移: 0x034

描述

QSPI 输出值设置

表 29.
GPIO_HI_OUT_SET
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子位设置操作，即 <code>GPIO_HI_OUT = wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_CLR 寄存器

偏移量: 0x038

描述

QSPI 输出值清除

表 30.
GPIO_HI_OUT_CLR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子位清除操作，即 <code>GPIO_HI_OUT &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OUT_XOR 寄存器

偏移量: 0x03c

描述

QSPI 输出值异或

表 31.
GPIO_HI_OUT_XOR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OUT 执行原子按位异或操作，即 <code>GPIO_HI_OUT ^= wdata</code>	WO	0x00

SIO: GPIO_HI_OE 寄存器

偏移量: 0x040

描述

QSPI 输出使能

表 32. GPIO_HI_OE
寄存器

位	描述	类型	复位值
31:6	保留。	-	-

Bits	Description	Type	Reset
5:0	Set output enable (1/0 → output/input) for QSPI IO0...5. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_HI_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00

SIO: GPIO_HI_OE_SET Register

Offset: 0x044

Description

QSPI output enable set

Table 33.
GPIO_HI_OE_SET
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bit-set on GPIO_HI_OE, i.e. <code>GPIO_HI_OE = wdata</code>	WO	0x00

SIO: GPIO_HI_OE_CLR Register

Offset: 0x048

Description

QSPI output enable clear

Table 34.
GPIO_HI_OE_CLR
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bit-clear on GPIO_HI_OE, i.e. <code>GPIO_HI_OE &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OE_XOR Register

Offset: 0x04c

Description

QSPI output enable XOR

Table 35.
GPIO_HI_OE_XOR
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Perform an atomic bitwise XOR on GPIO_HI_OE, i.e. <code>GPIO_HI_OE ^= wdata</code>	WO	0x00

SIO: FIFO_ST Register

Offset: 0x050

Description

Status register for inter-core FIFOs (mailboxes).

There is one FIFO in the core 0 → core 1 direction, and one core 1 → core 0. Both are 32 bits wide and 8 words deep.

Core 0 can see the read side of the 1→0 FIFO (RX), and the write side of 0→1 FIFO (TX).

Core 1 can see the read side of the 0→1 FIFO (RX), and the write side of 1→0 FIFO (TX).

The SIO IRQ for each core is the logical OR of the VLD, WOF and ROE fields of its FIFO_ST register.

位	描述	类型	复位值
5:0	设置 QSPI IO0...5 的输出使能 (1/0 → 输出/输入)。 读取操作返回最后写入的值。 当核心 0 和核心 1 同时写入 GPIO_HI_OE (或其 SET/CLR/XOR 别名) 时， 则结果相当于核心0的写入先执行， 随后核心1的写入应用于该中间结果。	读写	0x00

SIO: GPIO_HI_OE_SET 寄存器

偏移量: 0x044

描述

QSPI 输出使能设置

表 33.
GPIO_HI_OE_SET
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子位设置操作, 即 <code>GPIO_HI_OE = wdata</code>	WO	0x00

SIO: GPIO_HI_OE_CLR 寄存器

偏移: 0x048

描述

QSPI 输出使能清除

表 34.
GPIO_HI_OE_CLR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子位清除操作, 即 <code>GPIO_HI_OE &= ~wdata</code>	WO	0x00

SIO: GPIO_HI_OE_XOR 寄存器

偏移: 0x04c

描述

QSPI 输出使能 XOR

表 35.
GPIO_HI_OE_XOR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对 GPIO_HI_OE 执行原子按位异或操作, 即 <code>GPIO_HI_OE ^= wdata</code>	WO	0x00

SIO: FIFO_ST 寄存器

偏移: 0x050

描述

核间FIFO（邮箱）状态寄存器。

核心0到核心1方向有一个FIFO，核心1到核心0方向也有一个。两个FIFO均为32位宽，深度为8字。

核心0可访问 1→0 FIFO (RX) 的读端，以及 0→1 FIFO (TX) 的写端。

Core 1 可见 0→1 FIFO (RX) 的读取端，以及 1→0 FIFO (TX) 的写入端。

每个核心的 SIO IRQ 为其 FIFO_ST 寄存器中 VLD、WOF 和 ROE 字段的逻辑“或”运算结果。

Table 36. FIFO_ST Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	ROE : Sticky flag indicating the RX FIFO was read when empty. This read was ignored by the FIFO.	WC	0x0
2	WOF : Sticky flag indicating the TX FIFO was written when full. This write was ignored by the FIFO.	WC	0x0
1	RDY : Value is 1 if this core's TX FIFO is not full (i.e. if FIFO_WR is ready for more data)	RO	0x1
0	VLD : Value is 1 if this core's RX FIFO is not empty (i.e. if FIFO_RD is valid)	RO	0x0

SIO: FIFO_WR Register

Offset: 0x054

Table 37. FIFO_WR Register

Bits	Description	Type	Reset
31:0	Write access to this core's TX FIFO	WF	0x00000000

SIO: FIFO_RD Register

Offset: 0x058

Table 38. FIFO_RD Register

Bits	Description	Type	Reset
31:0	Read access to this core's RX FIFO	RF	-

SIO: SPINLOCK_ST Register

Offset: 0x05c

Table 39. SPINLOCK_ST Register

Bits	Description	Type	Reset
31:0	Spinlock state A bitmap containing the state of all 32 spinlocks (1=locked). Mainly intended for debugging.	RO	0x00000000

SIO: DIV_UDIVIDEND Register

Offset: 0x060

Table 40. DIV_UDIVIDEND Register

Bits	Description	Type	Reset
31:0	Divider unsigned dividend Write to the DIVIDEND operand of the divider, i.e. the p in p / q . Any operand write starts a new calculation. The results appear in QUOTIENT, REMAINDER. UDIVIDEND/SDIVIDEND are aliases of the same internal register. The U alias starts an unsigned calculation, and the S alias starts a signed calculation.	RW	0x00000000

SIO: DIV_UDIVISOR Register

Offset: 0x064

表 36. FIFO_ST
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ROE : 粘滞标志, 表示 RX FIFO 在空时被读取。该读取操作被 FIFO 忽略。	WC	0x0
2	WOF : 粘滞标志, 表示 TX FIFO 在满时被写入。该写入操作被 FIFO 忽略。	WC	0x0
1	RDY : 当该核心的 TX FIFO 未满 (即 FIFO_WR 准备接收更多数据) 时, 值为 1。	只读	0x1
0	VLD : 当该核心的 RX FIFO 非空 (即 FIFO_RD 有效) 时, 值为 1。	只读	0x0

SIO: FIFO_WR 寄存器

偏移: 0x054

表 37. FIFO_WR
寄存器

位	描述	类型	复位值
31:0	该核TX FIFO的写访问权限	WF	0x00000000

SIO: FIFO_RD 寄存器

偏移: 0x058

表 38. FIFO_RD
寄存器

位	描述	类型	复位值
31:0	该核RX FIFO的读访问权限	RF	-

位	描述	类型	复位值
31:0	自旋锁状态 包含所有32个自旋锁状态的位图 (1=已锁定)。 主要用于调试。	只读	0x00000000

SIO: SPINLOCK_ST 寄存器

偏移: 0x05c

表 39.
SPINLOCK_ST
寄存器

SIO: DIV_UDIVIDEND 寄存器

偏移: 0x060

表 40.
DIV_UDIVIDEND
寄存器

位	描述	类型	复位值
31:0	除法器无符号被除数 写入除法器的DIVIDEND操作数, 即 p / q 中的 p 。 任何操作数的写入均启动新的计算。结果显示在QUOTIENT和REMAINDER寄存器中。 UDIVIDEND/SDIVIDEND为同一内部寄存器的别名。U别名启动无符号计算, S别名启动有符号计算。	读写	0x00000000

SIO: DIV_UDIVISOR 寄存器

偏移量: 0x064

Table 41.
DIV_UDIVISOR
Register

Bits	Description	Type	Reset
31:0	Divider unsigned divisor Write to the DIVISOR operand of the divider, i.e. the q in p / q. Any operand write starts a new calculation. The results appear in QUOTIENT, REMAINDER. UDIVISOR/SDIVISOR are aliases of the same internal register. The U alias starts an unsigned calculation, and the S alias starts a signed calculation.	RW	0x00000000

SIO: DIV_SDIVIDEND Register

Offset: 0x068

Table 42.
DIV_SDIVIDEND
Register

Bits	Description	Type	Reset
31:0	Divider signed dividend The same as UDIVIDEND, but starts a signed calculation, rather than unsigned.	RW	0x00000000

SIO: DIV_SDIVISOR Register

Offset: 0x06C

Table 43.
DIV_SDIVISOR
Register

Bits	Description	Type	Reset
31:0	Divider signed divisor The same as UDIVISOR, but starts a signed calculation, rather than unsigned.	RW	0x00000000

SIO: DIV_QUOTIENT Register

Offset: 0x070

Table 44.
DIV_QUOTIENT
Register

Bits	Description	Type	Reset
31:0	Divider result quotient The result of DIVIDEND / DIVISOR (division). Contents undefined while CSR_READY is low. For signed calculations, QUOTIENT is negative when the signs of DIVIDEND and DIVISOR differ. This register can be written to directly, for context save/restore purposes. This halts any in-progress calculation and sets the CSR_READY and CSR_DIRTY flags. Reading from QUOTIENT clears the CSR_DIRTY flag, so should read results in the order REMAINDER, QUOTIENT if CSR_DIRTY is used.	RW	0x00000000

SIO: DIV_REMAINDER Register

Offset: 0x074

表41。
DIV_UDIVISOR
寄存器

位	描述	类型	复位值
31:0	除法器无符号除数 写入除法器的除数操作数，即除式中p / q中的q。 任何操作数的写入均启动新的计算。结果显示在QUOTIENT和REMAINDER寄存器中。 UDIVISOR/SDIVISOR为同一内部寄存器的别名。U别名启动 S别名启动有符号计算。	读写	0x00000000

SIO: DIV_SDIVIDEND 寄存器

偏移量: 0x068

表42。
DIV_SDIVIDEND
寄存器

位	描述	类型	复位值
31:0	除法器有符号被除数 与UDIVIDEND相同，但启动有符号运算，而非无符号。	读写	0x00000000

SIO: DIV_SDIVISOR 寄存器

偏移量: 0x06C

表43。
DIV_SDIVISOR
寄存器

位	描述	类型	复位值
31:0	除法器有符号除数 与UDIVISOR相同，但启动有符号运算，而非无符号。	读写	0x00000000

SIO: DIV_QUOTIENT 寄存器

偏移量: 0x070

表44。
DIV_QUOTIENT
寄存器

位	描述	类型	复位值
31:0	除法器结果商 结果为DIVIDEND / DIVISOR (除法)。当CSR_READY为低电平时，内容未定义。 对于有符号计算，DIVIDEND与DIVISOR符号不同时，QUOTIENT为负。 该寄存器可直接写入，用于上下文保存/恢复目的。该操作将中断任何正在进行的计算，并设置CSR_READY及CSR_DIRTY标志。 读取QUOTIENT会清除CSR_DIRTY标志，因此若使用CSR_DIRTY，应按REMAINDER、QUOTIENT的顺序读取结果。	读写	0x00000000

SIO: DIV_REMAINDER 寄存器

偏移量: 0x074

Table 45.
DIV_REMAINDER
Register

Bits	Description	Type	Reset
31:0	<p>Divider result remainder</p> <p>The result of DIVIDEND % DIVISOR (modulo). Contents undefined while CSR_READY is low.</p> <p>For signed calculations, REMAINDER is negative only when DIVIDEND is negative.</p> <p>This register can be written to directly, for context save/restore purposes. This halts any in-progress calculation and sets the CSR_READY and CSR_DIRTY flags.</p>	RW	0x00000000

SIO: DIV_CSR Register

Offset: 0x078

Description

Control and status register for divider.

Table 46. DIV_CSR
Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	<p>DIRTY: Changes to 1 when any register is written, and back to 0 when QUOTIENT is read.</p> <p>Software can use this flag to make save/restore more efficient (skip if not DIRTY).</p> <p>If the flag is used in this way, it's recommended to either read QUOTIENT only, or REMAINDER and then QUOTIENT, to prevent data loss on context switch.</p>	RO	0x0
0	<p>READY: Reads as 0 when a calculation is in progress, 1 otherwise.</p> <p>Writing an operand (xDIVIDEND, xDIVISOR) will immediately start a new calculation, no matter if one is already in progress.</p> <p>Writing to a result register will immediately terminate any in-progress calculation and set the READY and DIRTY flags.</p>	RO	0x1

SIO: INTERP0_ACCUM0 Register

Offset: 0x080

Table 47.
INTERP0_ACCUM0
Register

Bits	Description	Type	Reset
31:0	Read/write access to accumulator 0	RW	0x00000000

SIO: INTERP0_ACCUM1 Register

Offset: 0x084

Table 48.
INTERP0_ACCUM1
Register

Bits	Description	Type	Reset
31:0	Read/write access to accumulator 1	RW	0x00000000

SIO: INTERP0_BASE0 Register

Offset: 0x088

表45。
DIV_REMAINDER
寄存器

位	描述	类型	复位值
31:0	除法器结果余数 结果为DIVIDEND % DIVISOR (取模)。当CSR_READY为低电平时，内容未定义。 对于有符号计算，仅当DIVIDEND为负时，REMAINDER才为负。 该寄存器可直接写入，用于上下文保存/恢复目的。该操作将中断任何正在进行的计算，并设置CSR_READY及CSR_DIRTY标志。	读写	0x00000000

SIO: DIV_CSR寄存器

偏移量: 0x078

描述

除法器控制与状态寄存器。

表46. DIV_CSR寄
存器

位	描述	类型	复位值
31:2	保留。	-	-
1	DIRTY : 当任何寄存器被写入时变为1，读取QUOTIENT时恢复为0。 软件可利用此标志提升保存/恢复操作的效率（非DIRTY时可跳过）。 若以此方式使用该标志，建议仅读取QUOTIENT，或先读取REMAINDER后读取QUOTIENT，以防上下文切换时数据丢失。	只读	0x0
0	READY : 计算进行中时读为0，完成后读为1。 写入操作数 (xDIVIDEND, xDIVISOR) 将立即启动新计算， 无论是否已有计算正在执行。 写入结果寄存器会立即中止任何当前计算， 并将READY与DIRTY标志设置为有效。	只读	0x1

SIO: INTERP0_ACCUM0寄存器

偏移量: 0x080

表47.
INTERP0_ACCUM
0寄存器

位	描述	类型	复位值
31:0	累加器0的读写访问	读写	0x00000000

SIO: INTERP0_ACCUM1寄存器

偏移: 0x084

表 48。
INTERP0_ACCUM1
寄存器

位	描述	类型	复位值
31:0	累加器1的读写访问	读写	0x00000000

SIO: INTERP0_BASE0 寄存器

偏移: 0x088

Table 49.
INTERP0_BASE0
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE0 register.	RW	0x00000000

SIO: INTERP0_BASE1 Register

Offset: 0x08c

Table 50.
INTERP0_BASE1
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE1 register.	RW	0x00000000

SIO: INTERP0_BASE2 Register

Offset: 0x090

Table 51.
INTERP0_BASE2
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE2 register.	RW	0x00000000

SIO: INTERP0_POP_LANE0 Register

Offset: 0x094

Table 52.
INTERP0_POP_LANE0
Register

Bits	Description	Type	Reset
31:0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP0_POP_LANE1 Register

Offset: 0x098

Table 53.
INTERP0_POP_LANE1
Register

Bits	Description	Type	Reset
31:0	Read LANE1 result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP0_POP_FULL Register

Offset: 0x09c

Table 54.
INTERP0_POP_FULL
Register

Bits	Description	Type	Reset
31:0	Read FULL result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP0_PEEK_LANE0 Register

Offset: 0xa0

Table 55.
INTERP0_PEEK_LANE
0 Register

Bits	Description	Type	Reset
31:0	Read LANE0 result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP0_PEEK_LANE1 Register

Offset: 0xa4

表 49。
INTERP0_BASE0
寄存器

位	描述	类型	复位值
31:0	BASE0寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_BASE1 寄存器

偏移: 0x08c

表 50。
INTERP0_BASE1
寄存器

位	描述	类型	复位值
31:0	BASE1寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_BASE2 寄存器

偏移: 0x090

表 51。
INTERP0_BASE2
寄存器

位	描述	类型	复位值
31:0	BASE2寄存器的读写访问。	读写	0x00000000

SIO: INTERP0_POP_LANE0 寄存器

偏移: 0x094

表 52。
INTERP0_POP_LANE0
寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP0_POP_LANE1 寄存器

偏移: 0x098

表 53
INTERP0_POP_LANE1
寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP0_POP_FULL 寄存器

偏移: 0x09c

表 54
INTERP0_POP_FULL
寄存器

位	描述	类型	复位值
31:0	读取 FULL 结果，并同时将通道结果写入两个累加器 (POP)	只读	0x00000000

SIO: INTERP0_PEEK_LANE0 寄存器

偏移: 0x0a0

表 55
INTERP0_PEEK_LANE
0 寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP0_PEEK_LANE1 寄存器

偏移: 0x0a4

Table 56.
INTERP0_PEEK_LANE
1 Register

Bits	Description	Type	Reset
31:0	Read LANE1 result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP0_PEEK_FULL Register

Offset: 0x0a8

Table 57.
INTERP0_PEEK_FULL
Register

Bits	Description	Type	Reset
31:0	Read FULL result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP0_CTRL_LANE0 Register

Offset: 0x0ac

Description

Control register for lane 0

Table 58.
INTERP0_CTRL_LANE
0 Register

Bits	Description	Type	Reset
31:26	Reserved.	-	-
25	OVERF : Set if either OVERF0 or OVERF1 is set.	RO	0x0
24	OVERF1 : Indicates if any masked-off MSBs in ACCUM1 are set.	RO	0x0
23	OVERF0 : Indicates if any masked-off MSBs in ACCUM0 are set.	RO	0x0
22	Reserved.	-	-
21	BLEND : Only present on INTERP0 on each core. If BLEND mode is enabled: - LANE1 result is a linear interpolation between BASE0 and BASE1, controlled by the 8 LSBs of lane 1 shift and mask value (a fractional number between 0 and 255/256ths) - LANE0 result does not have BASE0 added (yields only the 8 LSBs of lane 1 shift+mask value) - FULL result does not have lane 1 shift+mask value added (BASE2 + lane 0 shift+mask) LANE1 SIGNED flag controls whether the interpolation is signed or unsigned.	RW	0x0
20:19	FORCE_MSB : ORed into bits 29:28 of the lane result presented to the processor on the bus. No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM.	RW	0x0
18	ADD_RAW : If 1, mask + shift is bypassed for LANE0 result. This does not affect FULL result.	RW	0x0
17	CROSS_RESULT : If 1, feed the opposite lane's result into this lane's accumulator on POP.	RW	0x0
16	CROSS_INPUT : If 1, feed the opposite lane's accumulator into this lane's shift + mask hardware. Takes effect even if ADD_RAW is set (the CROSS_INPUT mux is before the shift+mask bypass)	RW	0x0
15	SIGNED : If SIGNED is set, the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE0, and LANE0 PEEK/POP appear extended to 32 bits when read by processor.	RW	0x0

表 56
INTERPO_PEEK_LANE
1 寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERPO_PEEK_FULL 寄存器

偏移: 0x0a8

表57。
INTERPO_PEEK_FULL
寄存器

位	描述	类型	复位值
31:0	读取完整结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERPO_CTRL_LANE0 寄存器

偏移: 0x0ac

描述

LANE 0 控制寄存器

表58。
INTERPO_CTRL_LANE
0 寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25	OVERF: 如果 OVERF0 或 OVERF1 被置位，则该位被置位。	只读	0x0
24	OVERF1: 指示 ACCUM1 中任何被屏蔽的高有效位（MSBs）是否被置位。	只读	0x0
23	OVERF0: 指示 ACCUM0 中任何被屏蔽的高有效位（MSBs）是否被置位。	只读	0x0
22	保留。	-	-
21	BLEND: 仅存在于每个核心的 INTERPO 模块中。若启用 BLEND 模式： - LANE1 结果为 BASE0 与 BASE1 之间的线性插值，由 lane 1 的移位与掩码值的最低 8 位控制（介于 0 与 255/256 之间的分数值）。 - LANE0 结果不包含 BASE0（仅输出 lane 1 移位与掩码值的最低 8 位）。 - FULL 结果未加上车道 1 的位移+掩码值（BASE2 + 车道 0 位移+掩码） LANE1 SIGNED 标志用于控制插值的有符号或无符号属性。	读写	0x0
20:19	FORCE_MSB: 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW: 若设置为 1，则绕过 LANE0 结果的掩码与位移处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT: 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT: 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED: 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE0 之前，LANE0 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0

Bits	Description	Type	Reset
14:10	MASK_MSB : The most-significant bit allowed to pass by the mask (inclusive) Setting MSB < LSB may cause chip to turn inside-out	RW	0x00
9:5	MASK_LSB : The least-significant bit allowed to pass by the mask (inclusive)	RW	0x00
4:0	SHIFT : Logical right-shift applied to accumulator before masking	RW	0x00

SIO: INTERP0_CTRL_LANE1 Register

Offset: 0x0b0

Description

Control register for lane 1

Table 59.
INTERP0_CTRL_LANE1 Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20:19	FORCE_MSB : ORed into bits 29:28 of the lane result presented to the processor on the bus. No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM.	RW	0x0
18	ADD_RAW : If 1, mask + shift is bypassed for LANE1 result. This does not affect FULL result.	RW	0x0
17	CROSS_RESULT : If 1, feed the opposite lane's result into this lane's accumulator on POP.	RW	0x0
16	CROSS_INPUT : If 1, feed the opposite lane's accumulator into this lane's shift + mask hardware. Takes effect even if ADD_RAW is set (the CROSS_INPUT mux is before the shift+mask bypass)	RW	0x0
15	SIGNED : If SIGNED is set, the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE1, and LANE1 PEEK/POP appear extended to 32 bits when read by processor.	RW	0x0
14:10	MASK_MSB : The most-significant bit allowed to pass by the mask (inclusive) Setting MSB < LSB may cause chip to turn inside-out	RW	0x00
9:5	MASK_LSB : The least-significant bit allowed to pass by the mask (inclusive)	RW	0x00
4:0	SHIFT : Logical right-shift applied to accumulator before masking	RW	0x00

SIO: INTERP0_ACCUM0_ADD Register

Offset: 0x0b4

Table 60.
INTERP0_ACCUM0_ADD Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Values written here are atomically added to ACCUM0 Reading yields lane 0's raw shift and mask value (BASE0 not added).	RW	0x000000

SIO: INTERP0_ACCUM1_ADD Register

Offset: 0x0b8

位	描述	类型	复位值
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP0_CTRL_LANE1 寄存器

偏移量: 0x0b0

说明

LANE 1 控制寄存器

表 59。
INTERP0_CTRL_LANE
1 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若值为 1，LANE1 结果将绕过掩码和移位处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT : 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED : 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE1 之前，LANE1 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP0_ACCUM0_ADD 寄存器

偏移量: 0x0b4

表 60。
INTERP0_ACCUM0_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至 ACCUM0 读取结果为 lane 0 的原始位移及掩码值（未加 BASE0）。	读写	0x000000

SIO: INTERP0_ACCUM1_ADD 寄存器

偏移: 0x0b8

Table 61.
INTERP0_ACCUM1_AD
D Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Values written here are atomically added to ACCUM1 Reading yields lane 1's raw shift and mask value (BASE1 not added).	RW	0x000000

SIO: INTERP0_BASE_1AND0 Register

Offset: 0x0bc

Table 62.
INTERP0_BASE_1AND
0 Register

Bits	Description	Type	Reset
31:0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously. Each half is sign-extended to 32 bits if that lane's SIGNED flag is set.	WO	0x00000000

SIO: INTERP1_ACCUM0 Register

Offset: 0x0c0

Table 63.
INTERP1_ACCUM0
Register

Bits	Description	Type	Reset
31:0	Read/write access to accumulator 0	RW	0x00000000

SIO: INTERP1_ACCUM1 Register

Offset: 0x0c4

Table 64.
INTERP1_ACCUM1
Register

Bits	Description	Type	Reset
31:0	Read/write access to accumulator 1	RW	0x00000000

SIO: INTERP1_BASE0 Register

Offset: 0x0c8

Table 65.
INTERP1_BASE0
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE0 register.	RW	0x00000000

SIO: INTERP1_BASE1 Register

Offset: 0x0cc

Table 66.
INTERP1_BASE1
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE1 register.	RW	0x00000000

SIO: INTERP1_BASE2 Register

Offset: 0x0d0

Table 67.
INTERP1_BASE2
Register

Bits	Description	Type	Reset
31:0	Read/write access to BASE2 register.	RW	0x00000000

SIO: INTERP1_POP_LANE0 Register

Offset: 0x0d4

表 61。
INTERP0_ACCUM1_A
DD寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM1 读取返回通道1的原始位移和值掩码（未加BASE1）。	读写	0x000000

SIO: INTERP0_BASE_1AND0 寄存器

偏移: 0x0bc

表 62。
INTERP0_BASE_1AN
D0寄存器

位	描述	类型	复位值
31:0	写入时，低16位同时写入BASE0，高位写入BASE1。 若该通道的SIGNED标志被设置，则每半部分将符号扩展至32位。	WO	0x00000000

SIO: INTERP1_ACCUM0 寄存器

偏移: 0x0c0

表 63。
INTERP1_ACCUM
0寄存器

位	描述	类型	复位值
31:0	累加器0的读写访问	读写	0x00000000

SIO: INTERP1_ACCUM1 寄存器

偏移: 0x0c4

表 64。
INTERP1_ACCUM
1寄存器

位	描述	类型	复位值
31:0	累加器1的读写访问	读写	0x00000000

SIO: INTERP1_BASE0 寄存器

偏移: 0x0c8

表 65。
INTERP1_BASE0
寄存器

位	描述	类型	复位值
31:0	BASE0寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_BASE1 寄存器

偏移: 0x0cc

表 66。
INTERP1_BASE1
寄存器

位	描述	类型	复位值
31:0	BASE1寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_BASE2 寄存器

偏移: 0x0d0

表 67。
INTERP1_BASE2
寄存器

位	描述	类型	复位值
31:0	BASE2寄存器的读写访问。	读写	0x00000000

SIO: INTERP1_POP_LANE0 寄存器

偏移: 0x0d4

Table 68.
INTERP1_POP_LANE0
Register

Bits	Description	Type	Reset
31:0	Read LANE0 result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP1_POP_LANE1 Register

Offset: 0x0d8

Table 69.
INTERP1_POP_LANE1
Register

Bits	Description	Type	Reset
31:0	Read LANE1 result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP1_POP_FULL Register

Offset: 0x0dc

Table 70.
INTERP1_POP_FULL
Register

Bits	Description	Type	Reset
31:0	Read FULL result, and simultaneously write lane results to both accumulators (POP).	RO	0x00000000

SIO: INTERP1_PEEK_LANE0 Register

Offset: 0x0e0

Table 71.
INTERP1_PEEK_LANE
0 Register

Bits	Description	Type	Reset
31:0	Read LANE0 result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP1_PEEK_LANE1 Register

Offset: 0x0e4

Table 72.
INTERP1_PEEK_LANE
1 Register

Bits	Description	Type	Reset
31:0	Read LANE1 result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP1_PEEK_FULL Register

Offset: 0x0e8

Table 73.
INTERP1_PEEK_FULL
Register

Bits	Description	Type	Reset
31:0	Read FULL result, without altering any internal state (PEEK).	RO	0x00000000

SIO: INTERP1_CTRL_LANE0 Register

Offset: 0x0ec

Description

Control register for lane 0

Table 74.
INTERP1_CTRL_LANE
0 Register

Bits	Description	Type	Reset
31:26	Reserved.	-	-
25	OVERF : Set if either OVERF0 or OVERF1 is set.	RO	0x0
24	OVERF1 : Indicates if any masked-off MSBs in ACCUM1 are set.	RO	0x0
23	OVERF0 : Indicates if any masked-off MSBs in ACCUM0 are set.	RO	0x0

表 68。
INTERP1_POP_LANE0
寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP1_POP_LANE1 寄存器

偏移: 0x0d8

表 69。
INTERP1_POP_LANE1
寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，同时将通道结果写入两个累加器（弹出操作） 。	只读	0x00000000

SIO: INTERP1_POP_FULL 寄存器

偏移: 0x0dc

表 70。
INTERP1_POP_FULL
寄存器

位	描述	类型	复位值
31:0	读取 FULL 结果，并同时将通道结果写入两个累加器 (POP)	只读	0x00000000

SIO: INTERP1_PEEK_LANE0 寄存器

偏移: 0x0e0

表 71。
INTERP1_PEEK_LANE
0 寄存器

位	描述	类型	复位值
31:0	读取LANE0结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_PEEK_LANE1 寄存器

偏移: 0x0e4

表 72。
INTERP1_PEEK_LANE
1 寄存器

位	描述	类型	复位值
31:0	读取LANE1结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_PEEK_FULL 寄存器

偏移: 0x0e8

表 73。
INTERP1_PEEK_FULL
寄存器

位	描述	类型	复位值
31:0	读取完整结果，不改变任何内部状态（PEEK）。	只读	0x00000000

SIO: INTERP1_CTRL_LANE0 寄存器

偏移: 0x0ec

说明

LANE 0 控制寄存器

表 74
INTERP1_CTRL_LANE
0 寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25	OVERF : 如果 OVERF0 或 OVERF1 被置位，则该位被置位。	只读	0x0
24	OVERF1 : 指示 ACCUM1 中任何被屏蔽的高有效位 (MSBs) 是否被置位。	只读	0x0
23	OVERF0 : 指示 ACCUM0 中任何被屏蔽的高有效位 (MSBs) 是否被置位。	只读	0x0

Bits	Description	Type	Reset
22	CLAMP : Only present on INTERP1 on each core. If CLAMP mode is enabled: - LANE0 result is shifted and masked ACCUM0, clamped by a lower bound of BASE0 and an upper bound of BASE1. - Signedness of these comparisons is determined by LANE0_CTRL_SIGNED	RW	0x0
21	Reserved.	-	-
20:19	FORCE_MSB : ORed into bits 29:28 of the lane result presented to the processor on the bus. No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM.	RW	0x0
18	ADD_RAW : If 1, mask + shift is bypassed for LANE0 result. This does not affect FULL result.	RW	0x0
17	CROSS_RESULT : If 1, feed the opposite lane's result into this lane's accumulator on POP.	RW	0x0
16	CROSS_INPUT : If 1, feed the opposite lane's accumulator into this lane's shift + mask hardware. Takes effect even if ADD_RAW is set (the CROSS_INPUT mux is before the shift+mask bypass)	RW	0x0
15	SIGNED : If SIGNED is set, the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE0, and LANE0 PEEK/POP appear extended to 32 bits when read by processor.	RW	0x0
14:10	MASK_MSB : The most-significant bit allowed to pass by the mask (inclusive) Setting MSB < LSB may cause chip to turn inside-out	RW	0x00
9:5	MASK_LSB : The least-significant bit allowed to pass by the mask (inclusive)	RW	0x00
4:0	SHIFT : Logical right-shift applied to accumulator before masking	RW	0x00

SIO: INTERP1_CTRL_LANE1 Register

Offset: 0x0f0

Description

Control register for lane 1

Table 75.
INTERP1_CTRL_LANE1 Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20:19	FORCE_MSB : ORed into bits 29:28 of the lane result presented to the processor on the bus. No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM.	RW	0x0
18	ADD_RAW : If 1, mask + shift is bypassed for LANE1 result. This does not affect FULL result.	RW	0x0
17	CROSS_RESULT : If 1, feed the opposite lane's result into this lane's accumulator on POP.	RW	0x0

位	描述	类型	复位值
22	CLAMP : 仅存在于每个核心上的 INTERP1。如启用 CLAMP 模式: - LANE0 结果为移位并掩码处理后的 ACCUM0，受 BASE0 下限及 BASE1 上限约束。 - 这些比较的符号属性由 LANE0_CTRL_SIGNED 决定。	读写	0x0
21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若设置为 1，则绕过 LANE0 结果的掩码与位移处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0
16	CROSS_INPUT : 若设置为 1，将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW，亦会生效（CROSS_INPUT 多路复用器位于位移与掩码绕过之前）。	读写	0x0
15	SIGNED : 若设置 SIGNED，则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE0 之前，LANE0 的 PEEK/POP 显示扩展为 32 位 由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位（包含） 若设置 MSB 小于 LSB，可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位（包含）	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP1_CTRL_LANE1 寄存器

偏移: 0x0f0

描述

LANE 1 控制寄存器

表 75
INTERP1_CTRL_LANE
1 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:19	FORCE_MSB : 通过逻辑或作用于呈现给处理器总线的车道结果的第 29 至 28 位。 对内部 32 位数据通路无影响。适用于使用车道生成序列 指向闪存或 SRAM 的指针。	读写	0x0
18	ADD_RAW : 若值为 1，LANE1 结果将绕过掩码和移位处理。此设置不影响 FULL 结果。	读写	0x0
17	CROSS_RESULT : 若设置为 1，弹出操作时将对侧车道结果输入至此车道累加器。	读写	0x0

Bits	Description	Type	Reset
16	CROSS_INPUT : If 1, feed the opposite lane's accumulator into this lane's shift + mask hardware. Takes effect even if ADD_RAW is set (the CROSS_INPUT mux is before the shift+mask bypass)	RW	0x0
15	SIGNED : If SIGNED is set, the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE1, and LANE1 PEEK/POP appear extended to 32 bits when read by processor.	RW	0x0
14:10	MASK_MSB : The most-significant bit allowed to pass by the mask (inclusive) Setting MSB < LSB may cause chip to turn inside-out	RW	0x00
9:5	MASK_LSB : The least-significant bit allowed to pass by the mask (inclusive)	RW	0x00
4:0	SHIFT : Logical right-shift applied to accumulator before masking	RW	0x00

SIO: INTERP1_ACCUM0_ADD Register

Offset: 0x0f4

Table 76.
INTERP1_ACCUM0_AD
D Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Values written here are atomically added to ACCUM0 Reading yields lane 0's raw shift and mask value (BASE0 not added).	RW	0x000000

SIO: INTERP1_ACCUM1_ADD Register

Offset: 0x0f8

Table 77.
INTERP1_ACCUM1_AD
D Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Values written here are atomically added to ACCUM1 Reading yields lane 1's raw shift and mask value (BASE1 not added).	RW	0x000000

SIO: INTERP1_BASE_1AND0 Register

Offset: 0x0fc

Table 78.
INTERP1_BASE_1AND
0 Register

Bits	Description	Type	Reset
31:0	On write, the lower 16 bits go to BASE0, upper bits to BASE1 simultaneously. Each half is sign-extended to 32 bits if that lane's SIGNED flag is set.	WO	0x00000000

SIO: SPINLOCK0, SPINLOCK1, ..., SPINLOCK30, SPINLOCK31 Registers

Offsets: 0x100, 0x104, ..., 0x178, 0x17c

Table 79. SPINLOCK0,
SPINLOCK1, ...
SPINLOCK30,
SPINLOCK31
Registers

位	描述	类型	复位值
16	CROSS_INPUT : 若设置为 1, 将对侧车道累加器输入至此车道的位移与掩码硬件。 即使设置了 ADD_RAW, 亦会生效 (CROSS_INPUT 多路复用器位于位移与掩码绕过之前)。	读写	0x0
15	SIGNED : 若设置 SIGNED, 则位移并掩码后的累加器值会符号扩展至 32 位。 在添加到 BASE1 之前, LANE1 的 PEEK/POP 显示扩展为 32 位由处理器读取时。	读写	0x0
14:10	MASK_MSB : 掩码允许通过的最高有效位 (包含) 若设置 MSB 小于 LSB, 可能导致芯片功能异常	读写	0x00
9:5	MASK_LSB : 掩码允许通过的最低有效位 (包含)	读写	0x00
4:0	SHIFT : 掩码应用前对累加器执行的逻辑右移	读写	0x00

SIO: INTERP1_ACCUM0_ADD 寄存器

偏移: 0x0f4

表 76
INTERP1_ACCUM0_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM0 读取结果为 lane 0 的原始移位及掩码值 (未加 BASE0)。	读写	0x000000

SIO: INTERP1_ACCUM1_ADD 寄存器

偏移: 0x0f8

表 77。
INTERP1_ACCUM1_ADD
D 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	写入的值会以原子方式累加至ACCUM1 读取返回通道1的原始位移和值掩码 (未加BASE1)。	读写	0x000000

SIO: INTERP1_BASE_1AND0 寄存器

偏移: 0x0fc

表 78。
INTERP1_BASE_1AND0
D 寄存器

位	描述	类型	复位值
31:0	写入时, 低16位同时写入BASE0, 高位写入BASE1。 若该通道的SIGNED标志被设置, 则每半部分将符号扩展至32位。	WO	0x00000000

SIO: SPINLOCK0、SPINLOCK1、...、SPINLOCK30、SPINLOCK31 寄存器

偏移: 0x100、0x104、...、0x178、0x17c

表 79。SPINLOCK0、SPINLOCK1
、...、SPINLOCK30、SPINLOCK31
寄存器

Bits	Description	Type	Reset
31:0	<p>Reading from a spinlock address will:</p> <ul style="list-style-type: none"> - Return 0 if lock is already locked - Otherwise return nonzero, and simultaneously claim the lock <p>Writing (any value) releases the lock.</p> <p>If core 0 and core 1 attempt to claim the same lock simultaneously, core 0 wins.</p> <p>The value returned on success is $0x1 \ll \text{lock number}$.</p>	RW	0x00000000

2.3.2. Interrupts

Each core is equipped with a standard ARM Nested Vectored Interrupt Controller (NVIC) which has 32 interrupt inputs. Each NVIC has the same interrupts routed to it, with the exception of the GPIO interrupts: there is one GPIO interrupt per bank, per core. These are completely independent, so e.g. core 0 can be interrupted by GPIO 0 in bank 0, and core 1 by GPIO 1 in the same bank.

On RP2040, only the lower 26 IRQ signals are connected on the NVIC, and IRQs 26 to 31 are tied to zero (never firing). The core can still be forced to enter the relevant interrupt handler by writing bits 26 to 31 in the NVIC [ISPR](#) register.

Table 80. Interrupts

IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

ⓘ NOTE

[XIP_IRQ](#) is from the [SSI](#) block that makes up part of the [XIP](#) block. It could be used in a configuration where code is running from SRAM instead of flash. In this configuration, the XIP block could be used as a normal SSI peripheral.

Nested interrupts are supported in hardware: a lower-priority interrupt can be preempted by a higher-priority interrupt (or another exception e.g. HardFault), and the lower-priority interrupt will resume once higher-priority exceptions have completed. The priority order is determined by:

- First, the dynamic priority level configured per interrupt by the [NVIC_IPR0-7](#) registers. The Cortex-M0+ implements the two most significant bits of an 8-bit priority field, so four priority levels are available, and the numerically-lowest level (level 0) is the highest priority.
- Second, for interrupts with the same dynamic priority level, the lower-numbered IRQ has higher priority (using the IRQ numbers given in the table above).

Some care has gone into arranging the RP2040 interrupt table to give a sensible default priority ordering, but individual interrupts can be raised or lowered in priority, using [NVIC_IPR0](#) through [NVIC_IPR7](#), to suit a particular use case.

The 26 system IRQ signals are masked (NMI mask) and then ORed together creating the NMI signal for the core. The NMI mask for each core can be configured using [PROC0_NMI_MASK](#) and [PROC1_NMI_MASK](#) in the Syscfg register block. Each of these registers has one bit for each system interrupt, and the each core's NMI is asserted if a system interrupt is asserted **and** the corresponding NMI mask bit is set for that core.

位	描述	类型	复位值
31:0	<p>从自旋锁地址读取操作将：</p> <ul style="list-style-type: none"> - 若锁已被占用，返回0 - 否则返回非零值，并同时成功占用该锁 <p>写入（任何值）将释放该锁。</p> <p>若核心0与核心1同时尝试占用同一锁，核心0胜出。</p> <p>成功时返回的值为 $0x1 << \text{锁编号}$。</p>	读写	0x00000000

2.3.2. 中断

每个核心均配备了标准的ARM嵌套向量中断控制器（NVIC），具有32个中断输入。

每个NVIC所接收的中断相同，但GPIO中断例外：每个核心针对每个GPIO组各有一个GPIO中断。这些中断完全独立，例如核心0可被第0组的GPIO 0中断，而核心1可被同一组的GPIO 1中断。

在RP2040上，仅有低26个IRQ信号连接至NVIC，而IRQ 26至31固定为零（永不触发）。

通过向NVIC ISPR寄存器的第26至31位写入数据，核心仍可被强制进入相关中断处理程序。

表80. 中断

IRQ	中断源	IRQ	中断源	IRQ	中断源	IRQ	中断源	IRQ	中断源
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

① 注意

XIP_IRQ来源于组成XIP模块部分的SSI模块。该中断可用于代码从SRAM而非闪存运行的配置中。在此配置中，XIP块可用作普通SSI外设。

硬件支持嵌套中断：低优先级中断可被高优先级中断（或其他异常，例如 HardFault）抢占，低优先级中断将在高优先级异常处理完成后恢复执行。优先级顺序由以下因素决定：

- 首先，由 NVIC_IPR0-7 寄存器为每个中断配置的动态优先级级别决定。Cortex-M0+ 实现了8位优先级字段中最高的两位，因此共提供四个优先级级别，其中数值最低的级别（级别0）为最高优先级。
- 其次，对于动态优先级相同的中断，编号较低的 IRQ 具有更高优先级（依据上述表格中的 IRQ 编号）。

RP2040 中断表经过精心安排，提供合理的默认优先级顺序，但可通过 NVIC_IPR0 至 NVIC_IPR7 调整个别中断优先级，以满足特定使用需求。

26个系统IRQ信号被屏蔽（NMI屏蔽），然后通过逻辑或合并产生核心的NMI信号。每个核心的NMI屏蔽可通过Syscfg寄存器块中的PROC0_NMI_MASK和PROC1_NMI_MASK配置。这些寄存器的每个位对应一个系统中断，当系统中断被触发且对应核心的NMI屏蔽位被设置时，该核心的NMI信号即被触发。

CAUTION

If the watchdog is armed, and some bits are set on the core 1 NMI mask, the RESETS block (and hence Syscfg) should be included in the watchdog reset list. Otherwise, following a watchdog event, core 1 NMI may be asserted when the core enter the bootrom. It is safe for core 0 to take an NMI when entering the bootrom (the handler will clear the NMI mask).

2.3.3. Event Signals

The Cortex-M0+ can enter a sleep state until an "event" (or interrupt) takes place, using the `WFE` instruction. It can also generate events, using the `SEV` instruction. On RP2040 the event signals are cross-wired between the two processors, so that an event sent by one processor will be received on the other.

NOTE

The event flag is "sticky", so if both processors send an event (`SEV`) simultaneously, and then both go to sleep (`WFE`), they will both wake immediately, rather than getting stuck in a sleep state.

While in a `WFE` (or `WFI`) sleep state, the processor can shut off its internal clock gates, consuming much less power. When **both** processors are sleeping, and the DMA is inactive, RP2040 as a whole can enter a sleep state, disabling clocks on unused infrastructure such as the busfabric, and waking automatically when one of the processors wakes. See [Section 2.11.2](#).

2.3.4. Debug

The 2-wire Serial Wire Debug (SWD) port provides access to hardware and software debug features including:

- Loading firmware into SRAM or external flash memory
- Control of processor execution: run/halt, step, set breakpoints, other standard Arm debug functionality
- Access to processor architectural state
- Access to memory and memory-mapped IO via the system bus

The SWD bus is exposed on two dedicated pins and is immediately available after power-on.

NOTE

We recommend a max SWD frequency of 24MHz. This depends heavily on your setup. You may need to run much slower (1MHz) depending on the quality and length of your cables.

Debug access is via independent DAPs (one per core) attached to a shared multidrop SWD bus (SWD v2). Each DAP will only respond to debug commands if correctly addressed by a SWD `TARGETSEL` command; all others tristate their outputs. Additionally, a Rescue DP (see [Section 2.3.4.2](#)) is available which is connected to system control features. Default addresses of each debug port are given below:

- Core 0: `0x01002927`
- Core 1: `0x11002927`
- Rescue DP: `0xf1002927`

The Instance IDs (top 4 bits of ID above) can be changed via a sysconfig register which may be useful in a multichip application. However note that ID=0xf is reserved for the internal Rescue DP (see [Section 2.3.4.2](#)).

⚠ 注意

若看门狗已启动，且核心1的NMI屏蔽中设置了某些位，则RESETS模块（包括Syscfg）应被纳入看门狗复位列表。否则，在看门狗事件发生后，当核心进入bootrom时，核心1的NMI可能会被触发。核心0在进入bootrom时接受NMI是安全的（处理程序会清除NMI屏蔽位）。

2.3.3. 事件信号

Cortex-M0+可使用 **WFE** 指令进入睡眠状态，直至发生“事件”（或中断）。它还可以使用 **SEV** 指令生成事件。在RP2040上，事件信号在两个处理器之间交叉连接，使得一个处理器发送的事件能够被另一个处理器接收。

ℹ 注意

事件标志具有“粘性”，因此若两个处理器同时发送事件（**SEV**），随后都进入休眠状态（**WFE**），两者会立即被唤醒，而不会陷入休眠。

处于 **WFE**（或 **WFI**）休眠状态时，处理器可以关闭其内部时钟门控，从而显著降低功耗。当两个处理器均处于休眠，且DMA处于非活动状态时，RP2040整体可以进入休眠状态，关闭未使用的基础设施时钟（如总线结构），并在任一处理器唤醒时自动复位。详见第2.11.2节。

2.3.4. 调试

2线制串行线调试（SWD）端口提供对硬件和软件调试功能的访问，包括：

- 将固件加载到SRAM或外部闪存中
- 处理器执行控制：运行/暂停、单步、设置断点及其他标准Arm调试功能
- 访问处理器架构状态
- 通过系统总线访问内存及内存映射IO

SWD总线通过两个专用引脚暴露，通电后即可使用。

ℹ 注意

我们建议最大SWD频率为24MHz。此设置高度依赖于您的具体配置。根据电缆的质量和长度，您可能需要以更低频率（1MHz）运行。

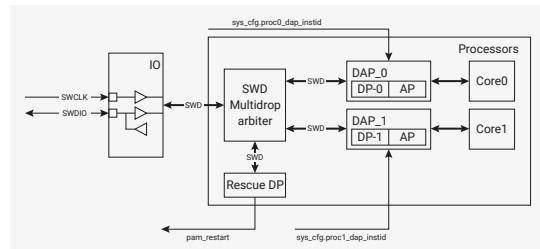
调试访问通过附着于共享多点SWD总线（SWD v2）的独立DAP（每核一个）实现。每个DAP仅在通过SWD **TARGETSEL**命令正确寻址后响应调试指令；所有其它DAP输出均处于高阻状态。

此外，提供一个Rescue DP（详见第2.3.4.2节），其连接至系统控制功能。各调试端口的默认地址如下：

- 核心 0: **0x01002927**
- 核心 1: **0x11002927**
- 救援 DP: **0xf1002927**

实例ID（上述ID的高4位）可通过sysconfig寄存器更改，这在多芯片应用中可能有所帮助。但请注意，ID=0xf保留用于内部救援DP（参见第2.3.4.2节）。

Figure 10. RP2040
Debugging



2.3.4.1. Software control of SWD pins

The SWD pins for Core 0 and Core 1 can be bit-banged via registers in syscfg (see [DBGFORCE](#)). This means that Core 1 could run a USB application that allows debug of Core 0, or similar.

2.3.4.2. Rescue DP

The Rescue DP (debug port) is available over the SWD bus and is only intended for use in the specific case where the chip has locked up, for example if code has been programmed into flash which permanently halts the system clock: in such a case, the normal debugger can not communicate with the processors to return the system to a working state, so more drastic action is needed. A [rescue](#) is invoked by setting the [CDBGWRUPREQ](#) bit in the Rescue DP's CTRL/STAT register.

This causes a hard reset of the chip (functionally similar to a power-on-reset), and sets a flag in the Chip Level Reset block to indicate that a rescue reset took place. The bootrom checks this flag almost immediately in the initial boot process (before watchdog, flash or USB boot), acknowledges by clearing the bit, then halts the processor. This leaves the system in a safe state, with the system clock running, so that the debugger can reattach to the cores and load fresh code.

For a practical example of using the Rescue DP, see the [Hardware design with RP2040](#) book.

2.4. Cortex-M0+

ARM Documentation

Excerpted from the [Cortex-M0+ Technical Reference Manual](#). Used with permission.

The ARM Cortex-M0+ processor is a very low gate count, highly energy efficient processor that is intended for microcontroller and deeply embedded applications that require an area optimized, low-power processor.

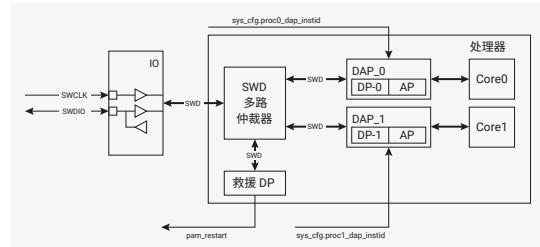
2.4.1. Features

The ARM Cortex-M0+ processor features and benefits are:

- Tight integration of system peripherals reduces area and development costs.
- Thumb instruction set combines high code density with 32-bit performance.
- Support for single-cycle I/O access.
- Power control optimization of system components.
- Integrated sleep modes for low-power consumption.
- Fast code execution enables running the processor with a slower clock or increasing sleep mode time.

图 10. RP2040

调试



2.3.4.1. SWD引脚的软件控制

Core 0和Core 1的SWD引脚可通过syscfg寄存器（参见DBGFORCE）进行位操作控制。这意味着Core 1可以运行一个USB应用程序，从而实现对Core 0的调试，或其他类似功能。

2.3.4.2. Rescue DP

Rescue DP（调试端口）通过SWD总线提供，仅用于芯片死锁的特定情况，例如代码被烧录至闪存且系统时钟永久停止：在此情形下，普通调试器无法与处理器通信以恢复系统至正常工作状态，因此需采取更为严厉的措施。通过在Rescue DP的CTRL/STAT寄存器中设置CDBGPWRUPREQ位来触发救援操作。

此操作将引起芯片硬复位（功能上类似于上电复位），并在芯片级复位模块中设置标志，指示救援复位已发生。bootrom在初始化启动过程中（看门狗、闪存或USB启动之前）几乎立即检查此标志，确认后通过清除该位进行响应，随后停止处理器。系统因此处于安全状态，系统时钟持续运行，便于调试器重新连接核心并加载新的代码。

有关使用Rescue DP的实用示例，请参见《Hardware design with RP2040》一书。

2.4. Cortex-M0+

ARM文档

摘自Cortex-M0+技术参考手册，已获授权使用。

ARM Cortex-M0+处理器为极低门数、高能效处理器，设计用于需面积优化及低功耗的微控制器和深度嵌入式应用。

2.4.1. 功能特性

ARM Cortex-M0+处理器的特性与优势包括：

- 系统外设的紧密集成降低了面积及开发成本。
- Thumb指令集兼具高代码密度与32位性能。
- 支持单周期I/O访问。
- 对系统组件的电源控制进行了优化。
- 集成了低功耗睡眠模式。
- 快速代码执行使处理器能够以较低的时钟频率运行或延长休眠模式时间。

- Optimized code fetching for reduced flash and ROM power consumption.
- Hardware multiplier.
- Deterministic, high-performance interrupt handling for time-critical applications.
- Deterministic instruction cycle timing.
- Support for system level debug authentication.
- Serial Wire Debug reduces the number of pins required for debugging.

2.4.1.1. Interfaces

The interfaces included in the processor for external access include:

- External AHB-Lite interface to busfabric
- Debug Access Port (DAP)
- Single-cycle I/O Port to SIO peripherals

2.4.1.2. Configuration

Each processor is configured with the following features:

- Architectural clock gating (for power saving)
- Little Endian bus access
- Four Breakpoints
- Debug support (via 2-wire debug pins [SWD/SWCLK](#))
- 32-bit instruction fetch (to match 32-bit data bus)
- IOPORT (for low latency access to local peripherals (see [SIO](#))
- 26 interrupts
- 8 MPU regions
- All registers reset on powerup
- Fast multiplier (MULS 32×32 single cycle)
- SysTick timer
- Vector Table Offset Register ([VTOR](#))
- 34 WIC (Wake-up Interrupt Controller) lines (32 IRQ and NMI, RXEV)
- DAP feature: Halt event support
- DAP feature: SerialWire debug interface (protocol 2 with multidrop support)
- DAP feature: Micro Trace Buffer (MTB) is not implemented

Architectural clock gating allows the processor core to support SLEEP and DEEPSLEEP power states by disabling the clock to parts of the processor core. Note that power gating is not supported.

Each M0+ core has its own interrupt controller which can individually mask out interrupt sources as required. The same interrupts are routed to both M0+ cores.

2.4.1.3. ARM architecture

The processor implements the ARMv6-M architecture profile. See the [ARMv6-M Architecture Reference Manual](#), and for

- 优化的代码取指以降低闪存和只读存储器的功耗。
- 硬件乘法器。
- 面向时间关键型应用的确定性高性能中断处理。
- 确定性的指令周期时序。
- 支持系统级调试认证。
- 串行线调试（Serial Wire Debug）减少调试所需的引脚数量。

2.4.1.1. 接口

处理器包含的用于外部访问的接口包括：

- 外部 AHB-Lite 接口至总线结构
- 调试访问端口（DAP）
- 单周期 I/O 端口至 SIO 外设

2.4.1.2. 配置

每个处理器配置有以下特性：

- 结构时钟门控（用于节能）
- 小端字节序总线访问
- 四个断点
- 调试支持（通过 2 线调试引脚 SWD/SWCLK）
- 32位指令取指（匹配32位数据总线）
- IOPORT（用于低延迟访问本地外设，详见SIO）
- 26个中断
- 8个MPU区域
- 所有寄存器上电复位
- 高速乘法器（MULS 32×32 单周期）
- SysTick定时器
- 向量表偏移寄存器（VTOR）
- 34条WIC（唤醒中断控制器）线路（32个IRQ和NMI，RXEV）
- DAP功能：支持停机事件
- DAP功能：SerialWire调试接口（协议2，支持多点连接）
- DAP功能：未实现微追踪缓冲区（MTB）

体系结构时钟门控允许处理器核心通过关闭部分核心时钟，支持SLEEP和DEEPSLEEP电源状态。请注意，不支持电源门控。

每个M0+核心都有独立的中断控制器，可根据需要单独屏蔽中断源。相同的中断信号被路由至两个M0+核心。

2.4.1.3. ARM 架构

该处理器实现了 ARMv6-M 架构概要。请参见《ARMv6-M 架构参考手册》，

further details refer to the [ARM Cortex M0+ Technical Reference Manual](#).

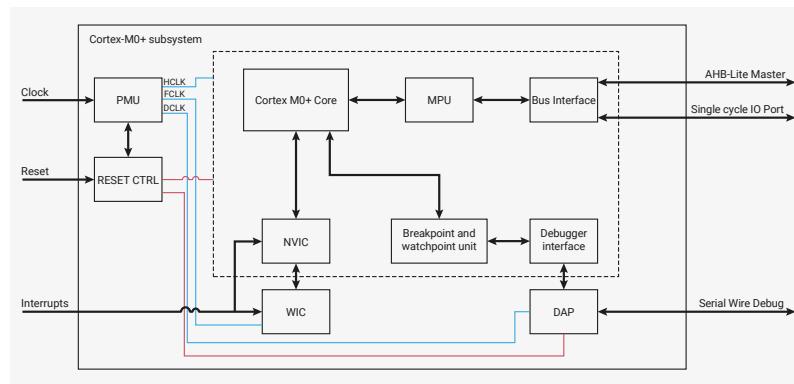
2.4.2. Functional Description

2.4.2.1. Overview

The Cortex-M0+ processor is a configurable, multistage, 32-bit RISC processor. It has an AMBA AHB-Lite interface and includes an NVIC component. It also has hardware debug, single-cycle I/O interfacing, and memory-protection functionality. The processor can execute Thumb code and is compatible with other Cortex-M profile processors.

[Figure 11](#) shows the functional blocks of the processor and surrounding blocks.

Figure 11. Cortex M0+ Functional block diagram



2.4.2.2. Features

The M0+ features:

- The ARMv6-M Thumb® instruction set.
- Thumb-2 technology.
- An ARMv6-M compliant 24-bit SysTick timer.
- A 32-bit hardware multiplier. This is the standard single-cycle multiplier
- The ability to have deterministic, fixed-latency, interrupt handling.
- Load/store multiple instructions that can be abandoned and restarted to facilitate rapid interrupt handling.
- C Application Binary Interface compliant exception model. This is the ARMv6-M, C Application Binary Interface (C-ABI) compliant exception model that enables the use of pure C functions as interrupt handlers.
- Low power sleep-mode entry using Wait For Interrupt (WFI), Wait For Event (WFE) instructions, or the return from interrupt sleep-on-exit feature.

2.4.2.3. NVIC features

The Nested Vectored Interrupt Controller (NVIC) features are:

- 26 external interrupt inputs, each with four levels of priority.
- Dedicated Non-Maskable Interrupt (NMI) input (which can be driven from any standard interrupt source)
- Support for both level-sensitive and pulse-sensitive interrupt lines.
- Wake-up Interrupt Controller (WIC), providing ultra-low power sleep mode support.
- Relocatable vector table.

更多详情请参考《ARM Cortex M0+ 技术参考手册》。

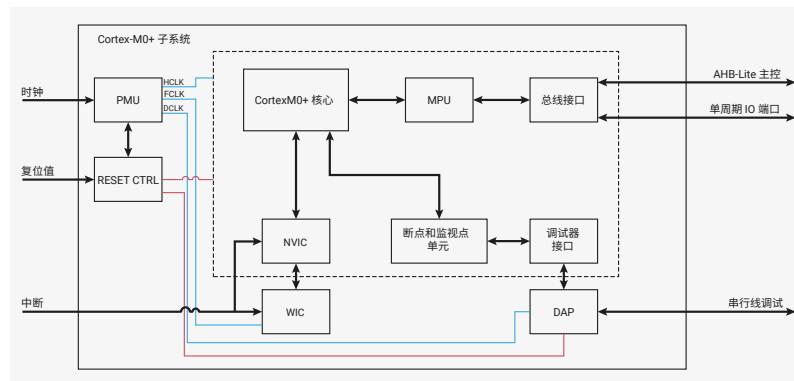
2.4.2. 功能描述

2.4.2.1. 概述

Cortex-M0+ 处理器是一款可配置的多级流水线 32 位 RISC 处理器。它具备 AMBA AHB-Lite 接口，并包含 NVIC 组件。同时还拥有硬件调试、单周期 I/O 接口及内存保护功能。该处理器能够执行 Thumb 代码，并与其他 Cortex-M 概要处理器兼容。

图 11 显示了处理器及其周边模块的功能模块。

图 11. Cortex M0+
功能模块
示意图



2.4.2.2. 特性

M0+ 的特性包括：

- ARMv6-M Thumb® 指令集。
- Thumb-2 技术。
- 符合 ARMv6-M 标准的 24 位 SysTick 计时器。
- 32 位硬件乘法器。这是标准的单周期乘法器。
- 具备确定性、固定延迟的中断处理能力。
- 支持可中止并可重启的多重加载/存储指令，以便实现快速中断处理。
- 符合 C 应用二进制接口的异常模型。这是 ARMv6-M 架构下符合 C 应用二进制接口（C-ABI）的异常模型，支持使用纯 C 函数作为中断处理程序。
- 采用 Wait For Interrupt (WFI)、Wait For Event (WFE) 指令，或中断返回时的睡眠退出功能，进入低功耗睡眠模式
-

2.4.2.3. NVIC 特性

嵌套向量中断控制器（NVIC）具备以下特性：

- 26 个外部中断输入，每个具备四个优先级等级。
- 专用的不可屏蔽中断 (NMI) 输入（可由任何标准中断源驱动）
- 支持电平触发和脉冲触发的中断线路。
- 唤醒中断控制器（WIC），实现超低功耗睡眠模式支持。
- 向量表可重定位。

NOTE

The NVIC supports hardware nesting of exceptions, e.g. an interrupt handler may itself be interrupted if a higher-priority interrupt request arrives whilst the handler is running.

Further details available in [Section 2.4.5](#).

2.4.2.4. Debug features

Debug features are:

- Four hardware breakpoints.
- Two watchpoints.
- Program Counter Sampling Register (PCSR) for non-intrusive code profiling.
- Single step and vector catch capabilities.
- Support for unlimited software breakpoints using BKPT instruction.
- Non-intrusive access to core peripherals and zero-waitstate system slaves through a compact bus matrix. A debugger can access these devices, including memory, even when the processor is running.
- Full access to core registers when the processor is halted.
- CoreSight compliant debug access through a Debug Access Port (DAP) supporting Serial Wire debug connections.

2.4.2.4.1. Debug Access Port

The processor is implemented with a low gate count Debug Access Port (DAP). The low gate count Debug Access Port (DAP) provides a Serial Wire debug-port, and connects to the processor slave port to provide full system-level debug access. For more information on DAP, see the ADI v5.1 version of the ARM Debug Interface v5, Architecture Specification

2.4.2.5. MPU features

Memory Protection Unit (MPU) features are:

- Eight user-configurable memory regions.
- Eight sub-region disables per region.
- Execute never (XN) support.
- Default memory map support.

Further details available in [Section 2.4.6](#).

2.4.2.6. AHB-Lite interface

Transactions on the AHB-Lite interface are always marked as non-sequential. Processor accesses and debug accesses share the external interface to external AHB peripherals. The processor accesses take priority over debug accesses. Any vendor-specific components can populate this bus.

注意

NVIC 支持硬件异常嵌套，例如当中断处理程序运行时，若有更高优先级中断请求到达，可中断当前处理程序。

详见第 2.4.5 节。

2.4.2.4. 调试功能

调试功能包括：

- 四个硬件断点。
- 两个监视点。
- 用于非侵入式代码分析的程序计数采样寄存器（PCSR）。
- 单步执行和向量捕获功能。
- 支持通过 BKPT 指令的无限制软件断点。
- 通过紧凑型总线矩阵对核心外设及零等待态系统从属设备进行非侵入式访问。调试器可在处理器运行时访问这些设备，包括内存。
- 处理器停止时，完全访问核心寄存器。
- 通过支持串行线调试连接的调试访问端口（DAP），实现符合 CoreSight 标准的调试访问。

2.4.2.4.1. 调试访问端口

处理器采用低门数调试访问端口（DAP）实现。低门数调试访问端口（DAP）提供串行线调试端口，并连接至处理器从属端口，提供完整的系统级调试访问。有关 DAP 的更多信息，请参见 ARM 调试接口 v5 架构规范的 ADI v5.1 版本。

2.4.2.5. MPU特性

内存保护单元（MPU）特性如下：

- 八个用户可配置的内存区域。
- 每个区域可禁用八个子区域。
- 支持执行禁止（XN）。
- 支持默认内存映射。

更多详细信息见第2.4.6节。

2.4.2.6. AHB-Lite接口

AHB-Lite接口上的传输始终标记为非连续。处理器访问与调试访问共享对外部AHB外设的外部接口。处理器访问优先于调试访问。

该总线可由任何厂商特定组件填充。

NOTE

Instructions are only fetched using the AHB-Lite interface. To optimize performance, the Cortex-M0+ processor fetches ahead of the instruction it is executing. To minimize power consumption, the fetch ahead is limited to a maximum of 32 bits.

2.4.2.7. Single-cycle I/O port

The processor implements a single-cycle I/O port that provides high speed access to tightly-coupled peripherals, such as general-purpose-I/O (GPIO). The port is accessible both by loads and stores from either the processor or the debugger. You cannot execute code from the I/O port.

2.4.2.8. Power Management Unit

Each processor has its own Power Management Unit (PMU) which allows power saving by turning off clocks to parts of the processor core. There are no separate power domains on RP2040.

The PMU runs from the processor clock which is controlled from the chip level clocks block. The PMU can control the following clock domains within the processor:

- A debug clock containing the processor debug resources and the rest of the DAP.
- A system clock containing the NVIC.
- A processor clock containing the core and associated interfaces

Control is limited to clock enable/disable. When enabled, all domains run at the same clock speed.

The PMU also interfaces with the WIC, to ensure that power-down and wake-up behaviours are transparent to software and work with clocking and sleeping requirements. This includes SLEEP or DEEPSLEEP support as controlled in [SCR](#) register.

2.4.2.8.1. Power Management

RP2040 ARM Cortex M0+ uses ARMv6-M which supports the use of Wait For Interrupt ([WFI](#)) and Wait For Event ([WFE](#)) instructions as part of system power management:

[WFI](#) provides a mechanism for hardware support of entry to one or more sleep states. Hardware can suspend execution until a wakeup event occurs.

[WFE](#) provides a mechanism for software to suspend program execution until a wakeup condition occurs with minimal or no impact on wakeup latency. Both [WFI](#) and [WFE](#) are hint instructions that might have no effect on program execution. Normally, they are used in software idle loops that resume program execution only after an interrupt or event of interest occurs.

NOTE

Code using [WFE](#) and [WFI](#) must handle any spurious wakeup events caused by a debug halt or other reasons.

Refer to the SDK and ARMv6-M guide for further information.

2.4.2.8.2. Wait For Event and Send Event

RP2040 can support software-based synchronization to system events using the Send-Event ([SEV](#)) and [WFE](#) hint instructions. Software can:

- use the [WFE](#) instruction to indicate that it is able to suspend execution of a process or thread until an event occurs, permitting hardware to enter a low power state.

注意

指令仅通过AHB-Lite接口抓取。为优化性能，Cortex-M0+处理器预取执行指令之后的指令。为降低功耗，预取长度限制为最多32位。

2.4.2.7. 单周期I/O端口

处理器实现单周期I/O端口，提供对紧耦合外设（如通用输入输出(GPIO)）的高速访问。该端口可由处理器或调试器进行读写访问。不得从I/O端口执行代码。

2.4.2.8. 电源管理单元

每个处理器均配备独立的电源管理单元（PMU），通过关闭处理器核心部分的时钟实现节能。RP2040上不存在独立的电源域。

PMU由处理器时钟驱动，该时钟由芯片级时钟模块控制。PMU可控制处理器内部以下时钟域：

- 包含处理器调试资源及其余DAP的调试时钟。
- 包含NVIC的系统时钟。
- 包含核心及相关接口的处理器时钟。

控制限于时钟的启用与禁用。启用时，所有时钟域均以相同频率运行。

PMU还通过WIC接口，确保掉电和唤醒行为对软件透明，且符合时钟及睡眠需求。包括由SCR寄存器控制的SLEEP或DEEP SLEEP支持。

2.4.2.8.1. 电源管理

RP2040 ARM Cortex M0+采用ARMv6-M架构，支持在系统电源管理中使用Wait For Interrupt ([WFI](#)) 和 Wait For Event ([WFE](#)) 指令：

[WFI](#)提供硬件支持进入一个或多个睡眠状态的机制。硬件可以挂起执行，直到唤醒事件发生为止。

[WFE](#)提供软件挂起程序执行的机制，直到唤醒条件出现，对唤醒延迟的影响极小或无影响。[WFI](#)和[WFE](#)均为提示指令，可能对程序执行无任何影响。

通常，这些指令用于软件空闲循环，仅在发生中断或相关事件后恢复程序执行。

注意

使用[WFE](#)和[WFI](#)的代码须能处理因调试暂停或其他原因引起的虚假唤醒事件。

详情请参阅SDK及ARMv6-M指南。

2.4.2.8.2. 等待事件与发送事件

RP2040可通过使用Send-Event ([SEV](#)) 和 [WFE](#)提示指令支持基于软件的系统事件同步。软件可以：

- 使用[WFE](#)指令表示能够挂起进程或线程的执行，直至事件发生，
允许硬件进入低功耗状态。

- rely on a mechanism that is transparent to software and provides low latency wakeup.

The **WFE** mechanism relies on hardware and software working together to achieve energy saving. For example, stalling execution of a processor until a device or another processor has set a flag:

- the hardware provides the mechanism to enter the **WFE** low-power state.
- software enters a polling loop to determine when the flag is set:
- the polling processor issues a **WFE** instruction as part of a polling loop if the flag is clear.
- an event is generated (hardware interrupt or Send-Event instruction from another processor) when the flag is set.

WFE wake up events

The following events are **WFE** wake up events:

- the execution of an **SEV** instruction on the other processor
- any exception entering the pending state if SEVONPEND in the System Control Register is set to 1.
- an asynchronous exception at a priority that preempts any currently active exceptions.
- a debug event with debug enabled.

The Event Register

The Event Register is a single bit register. When set, an Event Register indicates that an event has occurred, since the register was last cleared, that might prevent the processor having to suspend operation on issuing a **WFE** instruction. The following conditions apply to the Event Register:

- A reset clears the Event Register.
- Any **WFE** wakeup event, or the execution of an exception return instruction, sets the Event Register.
- A **WFE** instruction clears the Event Register.
- Software cannot read or write the value of the Event Register directly.

The Send-Event instruction

The Send-Event (**SEV**) instruction causes an event to be signalled to the other processor. The Send-Event instruction generates a wakeup event.

The Wait For Event instruction

The action of the **WFE** instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and returns immediately.
- If the Event Register is clear the processor can suspend execution and enter a low-power state. It can remain in that state until the processor detects a **WFE** wakeup event or a reset. When the processor detects a **WFE** wakeup event, the **WFE** instruction completes.

WFE wakeup events can occur before a **WFE** instruction is issued. Software using the **WFE** mechanism must tolerate spurious wake up events, including multiple wakeups.

2.4.2.8.3. Wait For Interrupt

RP2040 supports Wait For Interrupt through the hint instruction, **WFI**.

When a processor issues a **WFI** instruction it can suspend execution and enter a low-power state. It can remain in that state until the processor detects one of the following WFI wake up events:

- A reset.
- An asynchronous exception at a priority that, if PRIMASK.PM was set to 0, would preempt any currently active exceptions.

- 依赖对软件透明且具备低延迟唤醒能力的机制。

WFE机制依赖硬件与软件协同实现节能。例如，暂停处理器执行，直到设备或另一处理器设置标志：

- 硬件提供进入 WFE 低功耗状态的机制。
- 软件进入轮询循环以监测标志状态：
- 若标志未被设置，轮询处理器将在轮询循环中发出 WFE 指令。
- 当标志被设置时，会产生事件（硬件中断或来自另一处理器的 Send-Event 指令）。

WFE 唤醒事件

以下事件是 WFE 唤醒事件：

- 在另一处理器上执行 SEV 指令
- 若系统控制寄存器中的SEVONPEND位设置为1，则任何进入挂起状态的异常。
- 优先级高于当前活动异常的异步异常。
- 启用调试时发生的调试事件。

事件寄存器

事件寄存器为单比特寄存器。设置时，事件寄存器表示自上次清除以来发生了可能阻止处理器因执行 WFE 指令而挂起的事件。事件寄存器适用以下条件：

- 复位会清除事件寄存器。
- 任何 WFE 唤醒事件或异常返回指令的执行均会设置事件寄存器。
- 执行 WFE 指令会清除事件寄存器。
- 软件无法直接读写事件寄存器的值。

发送事件指令

发送事件（SEV）指令会向另一处理器发送事件信号。Send-Event 指令产生唤醒事件。

Wait For Event 指令

WFE 指令的操作依赖于事件寄存器的状态：

- 若事件寄存器被置位，该指令将清除寄存器并立即返回。
- 若事件寄存器未被置位，处理器可挂起执行并进入低功耗状态。处理器可持续处于该状态，直至检测到 WFE 唤醒事件或复位信号。当处理器检测到 WFE 唤醒事件时，WFE 指令完成执行。

WFE 唤醒事件可发生于 WFE 指令发出之前。应用 WFE 机制的软件必须能够容忍虚假唤醒事件，包括多次唤醒。

2.4.2.8.3. 等待中断

RP2040 通过提示指令 WFI 支持等待中断功能。

处理器发出 WFI 指令后，可挂起执行并进入低功耗状态。处理器可持续处于该状态，直至检测到以下任一 WFI 唤醒事件：

- 复位。
- 一个异步异常，其优先级若PRIMASK.PM设置为0，将抢占任何当前处于活动状态的异常。

Note

If `PRIMASK.PM` is set to 1, an asynchronous exception that has a higher group priority than any active exception results in a `WFI` instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.

- If debug is enabled, a debug event.
- A `WFI` wakeup event.

The `WFI` instruction completes when the hardware detects a `WFI` wake up event.

The processor recognizes `WFI` wake up events only after issuing the `WFI` instruction.

2.4.2.8.4. Wakeup Interrupt Controller

The Wakeup Interrupt Controller (WIC) is used to wake the processor from a DEEPSLEEP state as controlled by the `SCR` register. In a DEEPSLEEP state clocks to the processor core and NVIC are not running. It can take a few cycles to wake from a DEEPSLEEP state.

The WIC takes inputs from the receive event signal (from the other processor), 32 interrupts lines, and NMI.

For more power saving, RP2040 supports system level power saving modes as defined in [Section 2.11](#) which also includes code examples.

2.4.2.9. Reset Control

The Cortex M0+ Reset Control block controls the following resets:

- Debug reset
- M0+ core reset
- PMU reset

After power up, both processors are released from reset (see details in [Section 2.13.2](#)). This releases reset to Debug, M0+ core and PMU.

Once running, resets can be triggered from the Debugger, NVIC (using `AIRCR.SYSRESETREQ`), or the RP2040 Power On State Machine controller (see details in [Section 2.13](#)). The NVIC only resets the Cortex-M0+ processor core (not the Debug or PMU), whereas the Power On State Machine controller can reset the processor subsystem which asserts all resets in the subsystem (Debug, M0+ core, PMU).

2.4.3. Programmer's model

2.4.3.1. About the programmer's model

The ARMv6-M Architecture Reference Manual provides a complete description of the programmer's model. This chapter gives an overview of the Cortex-M0+ programmer's model that describes the implementation-defined options. It also contains the ARMv6-M Thumb instructions it uses and their cycle counts for the processor. Additional details are in following chapters

- [Section 2.4.4](#) summarizes the system control features of the programmer's model.
- [Section 2.4.5](#) summarizes the NVIC features of the programmer's model.
- [Section 2.3.4](#) summarizes the Debug features of the programmer's model.

注意

若 **PRIMASK.PM**设置为1，且异步异常的组优先级高于任何活动异常，则会导致 **WFI**指令退出。若异常的组优先级小于或等于执行组优先级，该异常将被忽略。

- 如果启用调试，则触发调试事件。
- 一个 **WFI**唤醒事件。

当硬件检测到**WFI**唤醒事件时，**WFI**指令即完成。

处理器仅在发出 **WFI** 指令后识别**WFI**唤醒事件。

2.4.2.8.4. 唤醒中断控制器

唤醒中断控制器（WIC）用于根据 **SCR**寄存器的控制，将处理器从DEEPSLEEP状态唤醒。在DEEPSLEEP状态下，处理器核心及NVIC的时钟处于停止状态。从深度睡眠（DEEPSLEEP）状态唤醒可能需要几个周期。

WIC 接收来自另一处理器的接收事件信号、32条中断线及非屏蔽中断（NMI）作为输入。

为了进一步节省功耗，RP2040 支持第2.11节定义的系统级节能模式，内附代码示例。

2.4.2.9. 复位控制

Cortex M0+ 复位控制模块管理以下复位：

- 调试复位
- M0+ 核心复位
- PMU 复位

上电后，两个处理器均解除复位（详见第2.13.2节），释放调试、M0+ 核心及 PMU 的复位信号。

运行时，复位可由调试器、NVIC（通过**AIRCR.SYSRESETREQ**）或 RP2040 上电状态机控制器触发（详见第2.13节）。NVIC 仅复位 Cortex-M0+ 处理器核心（不包括调试及 PMU），而上电状态机控制器则可复位处理器子系统，断言该子系统所有复位信号（调试、M0+ 核心及 PMU）。

2.4.3. 程序员模型

2.4.3.1 关于程序员模型

ARMv6-M 架构参考手册提供了程序员模型的完整描述。本章概述了Cortex-M0+程序员模型，说明了实现定义的选项。其中还包含该处理器所使用的ARMv6-M Thumb指令及其周期数。更多详细内容请参见后续章节。

- 第2.4.4节总结了程序员模型的系统控制功能。
- 第2.4.5节总结了程序员模型的NVIC功能。
- 第2.3.4节总结了程序员模型的调试功能。

2.4.3.2. Modes of operation and execution

See the ARMv6-M Architecture Reference Manual for information about the modes of operation and execution.

2.4.3.3. Instruction set summary

The processor implements the ARMv6-M Thumb instruction set, including a number of 32-bit instructions that use Thumb-2 technology. The ARMv6-M instruction set comprises:

- All of the 16-bit Thumb instructions from ARMv7-M excluding CBZ, CBNZ and IT.
- The 32-bit Thumb instructions BL, DMB, DSB, ISB, MRS and MSR.

[Table 81](#) shows the Cortex-M0+ instructions and their cycle counts. The cycle counts are based on a system with zero wait-states.

Table 81. Cortex-M0+ instruction summary

Operation	Description	Assembler	Cycles
Move	8-bit immediate	MOVS Rd, #<imm>	1
	Lo to Lo	MOVS Rd, Rm	1
	Any to Any	MOV Rd, Rm	1
	Any to PC	MOV PC, Rm	2
Add	3-bit immediate	ADDS Rd, Rn, #<imm>	1
	All registers Lo	ADDS Rd, Rn, Rm	1
	Any to Any	ADD Rd, Rd, Rm	1
	Any to PC	ADD PC, PC, Rm	2
	8-bit immediate	ADDS Rd, Rd, #<imm>	1
	With carry	ADCS Rd, Rd, Rm	1
	Immediate to SP	ADD SP, SP, #<imm>	1
	Form address from SP	ADD Rd, SP, #<imm>	1
	Form address from PC	ADR Rd, <label>	1
Subtract	Lo and Lo	SUBS Rd, Rn, Rm	1
	3-bit immediate	SUBS Rd, Rn, #<imm>	1
	8-bit immediate	SUBS Rd, Rd, #<imm>	1
	With carry	SBCS Rd, Rd, Rm	1
	Immediate from SP	SUB SP, SP, #<imm>	1
	Negate	RSBS Rd, Rn, #0	1
Multiply	Multiply	MULS Rd, Rm, Rd	1
Compare	Compare	CMP Rn, Rm	1
	Negative	CMN Rn, Rm	1
	Immediate	CMP Rn, #<imm>	1
Logical	AND	ANDS Rd, Rd, Rm	1
	Exclusive OR	EORS Rd, Rd, Rm	1
	OR	ORRS Rd, Rd, Rm	1

2.4.3.2 操作模式与执行

有关操作模式与执行的详情，请参阅ARMv6-M架构参考手册。

2.4.3.3 指令集概要

该处理器实现了ARMv6-M Thumb指令集，包括若干采用Thumb-2技术的32位指令。ARMv6-M 指令集包括：

- ARMv7-M 中除 CBZ、CBNZ 和 IT 以外的所有 16 位 Thumb 指令。
- 32 位 Thumb 指令：BL、DMB、DSB、ISB、MRS 和 MSR。

表 81 显示了 Cortex-M0+ 指令及其周期数。周期数基于零等待状态的系统。

表 81. Cortex-M0+
指令摘要

操作	描述	汇编指令	周期数
移动	8 位立即数	MOVS Rd, #<imm>	1
	低寄存器至低寄存器	MOVS Rd, Rm	1
	任意寄存器至任意寄存器	MOV Rd, Rm	1
	任意寄存器至 PC	MOV PC, Rm	2
加法	3位立即数	ADDS Rd, Rn, #<imm>	1
	所有寄存器 Lo	ADDS Rd, Rn, Rm	1
	任意寄存器至任意寄存器	ADD Rd, Rd, Rm	1
	任意寄存器至 PC	ADD PC, PC, Rm	2
减法	8 位立即数	ADDS Rd, Rd, #<imm>	1
	含进位	ADCS Rd, Rd, Rm	1
	立即数加至 SP	ADD SP, SP, #<imm>	1
	由 SP 生成地址	ADD Rd, SP, #<imm>	1
乘法	由 PC 生成地址	ADR Rd, <label>	1
	Lo 和 Lo	SUBS Rd, Rn, Rm	1
	3位立即数	SUBS Rd, Rn, #<imm>	1
	8 位立即数	SUBS Rd, Rd, #<imm>	1
比较	含进位	SBCS Rd, Rd, Rm	1
	来自 SP 的立即数	SUB SP, SP, #<imm>	1
	取负	RSBS Rd, Rn, #0	1
	乘法	MULS Rd, Rm, Rd	1
逻辑	比较	CMP Rn, Rm	1
	负值	CMN Rn, Rm	1
	立即数	CMP Rn, #<imm>	1
逻辑	与	ANDS Rd, Rd, Rm	1
	异或	EORS Rd, Rd, Rm	1
	或	ORRS Rd, Rd, Rm	1

Operation	Description	Assembler	Cycles
	Bit clear	BICS Rd, Rd, Rm	1
	Move NOT	MVNS Rd, Rm	1
	AND test	TST Rn, Rm	1
Shift	Logical shift left by immediate	LSLS Rd, Rm, #<shift>	1
	Logical shift left by register	LSLS Rd, Rd, Rs	1
	Logical shift right by immediate	LSRS Rd, Rm, #<shift>	1
	Logical shift right by register	LSRS Rd, Rd, Rs	1
	Arithmetic shift right	ASRS Rd, Rm, #<shift>	1
	Arithmetic shift right by register	ASRS Rd, Rd, Rs	1
Rotate	Rotate right by register	RORS Rd, Rd, Rs	1
Load	Word, immediate offset	LDR Rd, [Rn, #<imm>]	2 or 1 ^a
	Halfword, immediate offset	LDRH Rd, [Rn, #<imm>]	2 or 1 ^a
	Byte, immediate offset	LDRB Rd, [Rn, #<imm>]	2 or 1 ^a
	Word, register offset	LDR Rd, [Rn, Rm]	2 or 1 ^a
	Halfword, register offset	LDRH Rd, [Rn, Rm]	2 or 1 ^a
	Signed halfword, register offset	LDRSH Rd, [Rn, Rm]	2 or 1 ^a
	Byte, register offset	LDRB Rd, [Rn, Rm]	2 or 1 ^a
	Signed byte, register offset	LDRSB Rd, [Rn, Rm]	2 or 1 ^a
	PC-relative	LDR Rd, <label>	2 or 1 ^a
	SP-relative	LDR Rd, [SP, #<imm>]	2 or 1 ^a
	Multiple, excluding base	LDM Rn!, {<loreglist>}	1+N ^b
	Multiple, including base	LDM Rn, {<loreglist>}	1+N ^b
Store	Word, immediate offset	STR Rd, [Rn, #<imm>]	2 or 1 ^a
	Halfword, immediate offset	STRH Rd, [Rn, #<imm>]	2 or 1 ^a
	Byte, immediate offset	STRB Rd, [Rn, #<imm>]	2 or 1 ^a
	Word, register offset	STR Rd, [Rn, Rm]	2 or 1 ^a
	Halfword, register offset	STRH Rd, [Rn, Rm]	2 or 1 ^a
	Byte, register offset	STRB Rd, [Rn, Rm]	2 or 1 ^a
	SP-relative	STR Rd, [SP, #<imm>]	2 or 1 ^a
	Multiple	STM Rn!, {<loreglist>}	1+N ^b
Push	Push	PUSH {<loreglist>}	1+N ^b
	Push with link register	PUSH {<loreglist>, LR}	1+N ^c
Pop	Pop	POP {<loreglist>}	1+N ^b
	Pop and return	POP {<loreglist>, PC}	3+N ^c
Branch	Conditional	B<cc> <label>	1 or 2 ^d
	Unconditional	B <label>	2

操作	描述	汇编指令	周期数
	位清零	BICS Rd, Rd, Rm	1
	取反并移动	MVNS Rd, Rm	1
	与测试	TST Rn, Rm	1
移位操作	立即数逻辑左移	LSLS Rd, Rm, #<shift>	1
	寄存器逻辑左移	LSLS Rd, Rd, Rs	1
	立即数逻辑右移	LSRS Rd, Rm, #<shift>	1
	寄存器逻辑右移	LSRS Rd, Rd, Rs	1
	算术右移	ASRS Rd, Rm, #<shift>	1
	寄存器算术右移	ASRS Rd, Rd, Rs	1
旋转	寄存器右旋转	RORS Rd, Rd, Rs	1
加载	字, 立即偏移	LDR Rd, [Rn, #<imm>]	2 或 1 ^a
	半字, 立即偏移	LDRH Rd, [Rn, #<imm>]	2 或 1 ^a
	字节, 立即偏移	LDRB Rd, [Rn, #<imm>]	2 或 1 ^a
	字, 寄存器偏移	LDR Rd, [Rn, Rm]	2 或 1 ^a
	半字, 寄存器偏移	LDRH Rd, [Rn, Rm]	2 或 1 ^a
	带符号半字, 寄存器偏移	LDRSH Rd, [Rn, Rm]	2 或 1 ^a
	字节, 寄存器偏移	LDRB Rd, [Rn, Rm]	2 或 1 ^a
	带符号字节, 寄存器偏移	LDRSB Rd, [Rn, Rm]	2 或 1 ^a
	PC 相对	LDR Rd, <label>	2 或 1 ^a
	SP 相对	LDR Rd, [SP, #<imm>]	2 或 1 ^a
	多重载入, 排除基址	LDM Rn!, {<loreglist>}	1+N ^b
	多重载入, 包括基址	LDM Rn, {<loreglist>}	1+N ^b
存储	字, 立即偏移	STR Rd, [Rn, #<imm>]	2 或 1 ^a
	半字, 立即偏移	STRH Rd, [Rn, #<imm>]	2 或 1 ^a
	字节, 立即偏移	STRB Rd, [Rn, #<imm>]	2 或 1 ^a
	字, 寄存器偏移	STR Rd, [Rn, Rm]	2 或 1 ^a
	半字, 寄存器偏移	STRH Rd, [Rn, Rm]	2 或 1 ^a
	字节, 寄存器偏移	STRB Rd, [Rn, Rm]	2 或 1 ^a
	SP 相对	STR Rd, [SP, #<imm>]	2 或 1 ^a
	多重	STM Rn!, {<loreglist>}	1+N ^b
压入	压入	PUSH {<loreglist>}	1+N ^b
	带链接寄存器的压入	PUSH {<loreglist>, LR}	1+N ^c
弹出	弹出	POP {<loreglist>}	1+N ^b
	弹出并返回	POP {<loreglist>, PC}	3+N ^c
分支	条件分支	B<cc> <label>	1 或 2 ^d
	无条件分支	B <label>	2

Operation	Description	Assembler	Cycles
	With link	BL <label>	3
	With exchange	BX Rm	2
	With link and exchange	BLX Rm	2
Extend	Signed halfword to word	SXTB Rd, Rm	1
	Signed byte to word	SXTB Rd, Rm	1
	Unsigned halfword	UXTH Rd, Rm	1
	Unsigned byte	UXTB Rd, Rm	1
Reverse	Bytes in word	REV Rd, Rm	1
	Bytes in both halfwords	REV16 Rd, Rm	1
	Signed bottom half word	REVSH Rd, Rm	1
State	change Supervisor Call	SVC #<imm>	- e
	Disable interrupts	CPSID i	1
	Enable interrupts	CPSIE i	1
	Read special register	MRS Rd, <specreg>	3
	Write special register	MSR <specreg>, Rn	3
	Breakpoint	BKPT #<imm>	- e
Hint	Send-Event	SEV	1
	Wait For Event	WFE	2 ^f
	Wait For Interrupt	WFI	2 ^f
	Yield	YIELD	1 ^f
	No operation	NOP	1
Barriers	Instruction synchronization	ISB	3
	Data memory	DMB	3
	Data synchronization	DSB	3

Table Notes^a 2 if to AHB interface or SCS, 1 if to single-cycle I/O port.^b N is the number of elements in the list.^c N is the number of elements in the list including PC or LR.^d 2 if taken, 1 if not-taken.^e Cycle count depends on processor and debug configuration.^f Excludes time spent waiting for an interrupt or event.^g Executes as NOP.

See the ARMv6-M Architecture Reference Manual for more information about the ARMv6-M Thumb instructions.

2.4.3.4. Memory model

The processor contains a bus matrix that arbitrates the processor core and Debug Access Port (DAP) memory accesses to both the external memory system and to the internal NVIC and debug components.

Priority is always given to the processor to ensure that any debug accesses are as non-intrusive as possible. For a zero

操作	描述	汇编指令	周期数
	带链接	BL <label>	3
	带交换	BX Rm	2
	带链接和交换	BLX Rm	2
扩展	有符号半字扩展至字	SXTB Rd, Rm	1
	有符号字节扩展至字	SXTB Rd, Rm	1
	无符号半字	UXTH Rd, Rm	1
	无符号字节	UXTB Rd, Rm	1
字节反转	字中的字节	REV Rd, Rm	1
	两个半字中的字节	REV16 Rd, Rm	1
	有符号下半字	REVSH Rd, Rm	1
状态	更改主管调用 (SVC)	SVC #<imm>	- e
	禁用中断	CPSID i	1
	使能中断	CPSIE i	1
	读取特殊寄存器	MRS Rd, <specreg>	3
	写入特殊寄存器	MSR <specreg>, Rn	3
	断点	BKPT #<imm>	- e
提示	发送事件	SEV	1
	等待事件	WFE	2 ^f
	等待中断	WFI	2 ^f
	让出	YIELD	1 ^f
	无操作	NOP	1
屏障	指令同步	ISB	3
	数据存储器	DMB	3
	数据同步	DSB	3

表格注释

^a 当连接至AHB接口或SCS时为2，连接至单周期I/O端口时为1。

^b N为列表中元素的数量。

^c N为包含PC或LR的列表中元素数量。

^d 跳转时为2，不跳转时为1。

^e 周期数取决于处理器及调试配置。

^f 不计入等待中断或事件的时间。

^g 作为NOP指令执行。

有关ARMv6-M Thumb指令的详细信息，请参阅《ARMv6-M架构参考手册》。

2.4.3.4. 内存模型

该处理器包含一个总线矩阵，用于仲裁处理器核心和调试访问端口（DAP）对外部存储系统以及内部NVIC和调试组件的内存访问。

优先权始终赋予处理器，以确保所有调试访问尽可能不具侵入性。对于零

wait-state system, all debug accesses to system memory, NVIC, and debug resources are completely non-intrusive for typical code execution.

The system memory map is ARMv6-M architecture compliant, and is common both to the debugger and processor accesses. Transactions are routed as follows:

- All accesses below `0xd0000000` or above `0xefffffff` appear as AHB-Lite transactions on the AHB-Lite master port of the processor.
- Accesses in the range `0xd0000000` to `0xdfffffff` are handled by the SIO.
- Accesses in the range `0xe0000000` to `0xefffffff` are handled within the processor and do not appear on the AHB-Lite master port of the processor.

The processor supports only word size accesses in the range `0xd0000000` - `0xefffffff`.

[Table 82](#) shows the code, data, and device suitability for each region of the default memory map. This is the memory map used by implementations when the MPU is disabled. The attributes and permissions of all regions, except that targeting the Cortex-M0+ NVIC and debug components, can be modified using an implemented MPU.

Table 82. M0+ Default memory map usage

Address range	Code	Data	Device
<code>0xf0000000</code> - <code>0xffffffff</code>	No	No	Yes
<code>0xe0000000</code> - <code>0xffffffff</code>	No	No	No ^a
<code>0xa0000000</code> - <code>0xffffffff</code>	No	No	Yes
<code>0x60000000</code> - <code>0x9fffffff</code>	Yes	Yes	No
<code>0x40000000</code> - <code>0x5fffffff</code>	No	No	Yes
<code>0x20000000</code> - <code>0x3fffffff</code>	Yes	Yes	No
<code>0x00000000</code> - <code>0x1fffffff</code>	Yes	Yes	No

^a. Space reserved for Cortex-M0+ NVIC and debug components.

Note

Regions not marked as suitable for code behave as eXecute-Never (XN) and generate a HardFault exception if code attempts to execute from this location.

See the ARMv6-M Architecture Reference Manual for more information about the memory model.

2.4.3.5. Processor core registers summary

[Table 83](#) shows the processor core register set summary. Each of these registers is 32 bits wide.

Table 83. M0+ processor core register set summary

Name	Description
R0-R12	R0-R12 are general-purpose registers for data operations.
MSP/PSP (R13)	The Stack Pointer (SP) is register R13. In Thread mode, the CONTROL register indicates the stack pointer to use, Main Stack Pointer (MSP) or Process Stack Pointer (PSP).
LR (R14)	The Link Register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.
PC (R15)	The Program Counter (PC) is register R15. It contains the current program address.

等待状态系统，所有对系统内存、NVIC及调试资源的调试访问在典型代码执行时均完全不具侵入性。

系统内存映射符合ARMv6-M架构规范，且对调试器与处理器的访问均通用。事务路由如下：

- 所有低于 `0xd0000000` 或高于 `0xefffffff` 的访问，均作为 AHB-Lite 事务出现在处理器的 AHB-Lite 主端口。
- 位于范围 `0xd0000000` 至 `0xdfffffff` 的访问由 SIO 处理。
- 访问范围 `0xe0000000` 至 `0xefffffff` 在处理器内部处理，不会出现在处理器的 AHB-Lite 主端口。

处理器仅支持范围 `0xd` 内的字 (word) 大小访问 `00000000-0xefffffff`。

表82显示了默认内存映射中每个区域的代码、数据和设备适用性。此内存映射在禁用 MPU 时使用。除针对 Cortex-M0+ NVIC 及调试组件的区域外，所有区域的属性和权限均可通过已实现的 MPU 进行修改。

表82. M0+默认内存映射使用情况

地址范围	代码	数据	设备
<code>0xf0000000 - 0xffffffff</code>	否	否	是
<code>0xe0000000 - 0xefffffff</code>	否	否	否 ^a
<code>0xa0000000 - 0xdfffffff</code>	否	否	是
<code>0x60000000 - 0x9fffffff</code>	是	是	否
<code>0x40000000 - 0x5fffffff</code>	否	否	是
<code>0x20000000 - 0x3fffffff</code>	是	是	否
<code>0x00000000 - 0x1fffffff</code>	是	是	否

^a. 为 Cortex-M0+ NVIC 及调试组件预留的空间。

注意

未标记为可执行代码的区域视为禁止执行 (eXecute-Never, XN)；若代码尝试从该位置执行，将引发HardFault异常。

有关内存模型的详细信息，请参阅ARMv6-M架构参考手册。

2.4.3.5 处理器核寄存器摘要

表83列示了处理器核寄存器集合的摘要。每个寄存器宽度均为32位。

表83. M0+处理器核寄存器集合摘要

名称	描述
R0-R12	R0-R12为用于数据操作的通用寄存器。
MSP/PSP (R13)	堆栈指针 (SP) 是寄存器R13。在线程模式下，CONTROL寄存器指示使用的堆栈指针，即主堆栈指针 (MSP) 或进程堆栈指针 (PSP)。
LR (R14)	链接寄存器 (LR) 是寄存器R14。它存储子程序、函数调用及异常的返回地址。
PC (R15)	程序计数器 (PC) 是寄存器R15，包含当前程序的地址。

Name	Description
PSR	The Program Status Register (PSR) combines: <ul style="list-style-type: none"> Application Program Status Register (APSR). Interrupt Program Status Register (IPSR). Execution Program Status Register (EPSR). These registers provide different views of the PSR.
PRIMASK	The PRIMASK register prevents activation of all exceptions with configurable priority.
CONTROL	The CONTROL register controls the stack used, the code privilege level, when the processor is in Thread mode.

Note

See the ARMv6-M Architecture Reference Manual for information about the processor core registers and their addresses, access types, and reset values.

2.4.3.6. Exceptions

This section describes the exception model of the processor.

2.4.3.6.1. Exception handling

The processor implements advanced exception and interrupt handling, as described in the ARMv6-M Architecture Reference Manual. To minimize interrupt latency, the processor abandons any load-multiple or store-multiple instruction to take any pending interrupt. On return from the interrupt handler, the processor restarts the load-multiple or store-multiple instruction from the beginning.

This means that software must not use load-multiple or store-multiple instructions when a device is accessed in a memory region that is read-sensitive or sensitive to repeated writes. The software must not use these instructions in any case where repeated reads or writes might cause inconsistent results or unwanted side-effects.

The processor implementation can ensure that a fixed number of cycles are required for the NVIC to detect an interrupt signal and the processor fetch the first instruction of the associated interrupt handler. If this is done, the highest priority interrupt is jitter-free. This will depend on where the interrupt handler is located and if another higher priority master is accessing that memory. SRAM4 and SRAM5 are provided that may be allocated to interrupt handlers for each processor so this is jitter-free.

To reduce interrupt latency and jitter, the Cortex-M0+ processor implements both interrupt late-arrival and interrupt tail-chaining mechanisms, as defined by the ARMv6-M architecture. The worst case interrupt latency, for the highest priority active interrupt in a zero wait-state system not using jitter suppression, is 15 cycles.

The processor exception model has the following implementation-defined behaviour in addition to the architecture specified behaviour:

- Exceptions on stacking from HardFault to NMI lockup at NMI priority.
- Exceptions on unstacking from NMI to HardFault lockup at HardFault priority.

2.4.4. System control

名称	描述
PSR	程序状态寄存器（PSR）包含： <ul style="list-style-type: none">• 应用程序状态寄存器（APSR）• 中断程序状态寄存器（IPSR）• 执行程序状态寄存器（EPSR） 这些寄存器分别提供了对PSR的不同视图。
PRIMASK	PRIMASK寄存器禁止所有具有可配置优先级的异常被激活。
CONTROL	CONTROL寄存器在处理器处于线程模式时，控制所使用的堆栈及代码权限级别。

注意

有关处理器核心寄存器的地址、访问类型及复位值等信息，请参阅ARMv6-M架构参考手册。

2.4.3.6. 异常

本节介绍处理器的异常模型。

2.4.3.6.1. 异常处理

处理器实现了高级异常和中断处理，如《ARMv6-M 架构参考手册》中所述。为最小化中断延迟，处理器在响应任何待处理中断时，会放弃正在执行的多重加载（load-multiple）或多重存储（store-multiple）指令。从中断处理程序返回时，处理器将从头重新执行该多重加载或多重存储指令。

这意味着软件在访问只读敏感或对重复写入敏感的内存区域中的设备时，不得使用多重加载或多重存储指令。在任何可能导致不一致结果或不良副作用的重复读取或写入情况下，软件均不得使用此类指令。

处理器设计保证NVIC检测中断信号并使处理器取指向相关中断处理程序首条指令所需的周期数为固定值。如此一来，最高优先级中断将实现无抖动响应。这将取决于中断处理程序的位置以及是否有其他更高优先级的主控访问该内存。提供了SRAM4和SRAM5，可分配给各处理器的中断处理程序，因此无抖动。

为了减少中断延迟和抖动，Cortex-M0+处理器实现了ARMv6-M架构定义的中断后到达和中断尾部链接机制。在零等待状态系统中且未采用抖动抑制的情况下，最高优先级激活中断的最坏中断延迟为15个时钟周期。

处理器异常模型除架构规定的行为外，还具有以下实现定义行为：

- 从HardFault到NMI的堆栈异常将在NMI优先级下锁死。
- 从NMI到HardFault的弹栈异常将在HardFault优先级下锁死。

2.4.4. 系统控制

2.4.4.1. System control register summary

Table 84 gives the system control registers. Each of these registers is 32 bits wide.

Table 84. M0+ System control registers

Name	Description
SYST_CSR	SysTick Control and Status Register
SYST_RVR	SysTick Reload Value Register
SYST_CVR	SysTick Current Value Register
SYST_CALIB	SysTick Calibration value Register
CPUID	See CPUID Register
ICSR	Interrupt Control State Register
AIRCR	Application Interrupt and Reset Control Register
CCR	Configuration and Control Register
SHPR2	System Handler Priority Register
SHPR3	System Handler Priority Register
SHCSR	System Handler Control and State Register
VTOR	Vector table Offset Register
ACTLR	Auxiliary Control Register

Note

- All system control registers are only accessible using word transfers. Any attempt to read or write a halfword or byte is Unpredictable.
- See the List of Registers or ARMv6-M Architecture Reference Manual for more information about the system control registers, and their addresses and access types, and reset values.

2.4.4.1.1. CPUID Register

The [CPUID](#) contains the part number, version, and implementation information that is specific to the processor.

⚠️ IMPORTANT

This standard internal Arm register contains information about the type of processor. It should not be confused with [CPUID \(Section 2.3.1.1\)](#), an RP2040 SIO register which reads as 0 on core 0 and 1 on core 1.

2.4.5. NVIC

2.4.5.1. About the NVIC

External interrupt signals connect to the Nested Vectored Interrupt Controller (NVIC), and the NVIC prioritizes the interrupts. Software can set the priority of each interrupt. The NVIC and the Cortex-M0+ processor core are closely coupled, providing low latency interrupt processing and efficient processing of late arriving interrupts.

2.4.4.1 系统控制寄存器概述

表84列出了系统控制寄存器。每个寄存器宽度均为32位。

表84。M0+ 系统控制寄存器

名称	描述
SYST_CSR	SysTick控制与状态寄存器
SYST_RVR	SysTick 重装载值寄存器
SYST_CVR	SysTick 当前值寄存器
SYST_CALIB	SysTick 校准值寄存器
CPUID	参见 CPUID 寄存器
ICSR	中断控制和状态寄存器
AIRCR	应用程序中断及复位控制寄存器
CCR	配置与控制寄存器
SHPR2	系统处理优先级寄存器
SHPR3	系统处理优先级寄存器
SHCSR	系统处理控制和状态寄存器
VTOR	向量表偏移寄存器
ACTLR	辅助控制寄存器

注意

- 所有系统控制寄存器仅能通过字传输进行访问。任何尝试读取或写入半字或字节的操作均属不可预测。
- 有关系统控制寄存器及其地址、访问类型和复位值的详细信息，请参阅《寄存器列表》或《ARMv6-M体系结构参考手册》。

2.4.4.1.1. CPUID寄存器

CPUID寄存器包含特定于处理器的部件编号、版本和实现信息。

！重要

该标准内部Arm寄存器包含有关处理器类型的信息。其不应与CPUID（第2.3.1.1节）混淆，后者为RP2040 S10寄存器，在核心0上读取值为0，在核心1上读取值为1。

2.4.5. NVIC

2.4.5.1. 关于NVIC

外部中断信号连接至嵌套向量中断控制器（NVIC），NVIC负责中断的优先级排序。软件可以设置各中断的优先级。NVIC与Cortex-M0+处理器核心紧密耦合，提供低延迟的中断处理及对后发中断的高效处理能力。

NOTE

"Nested" refers to the fact that interrupts can themselves be interrupted, by higher-priority interrupts. "Vectored" refers to the hardware dispatching each interrupt to a distinct handler routine, specified by the vector table. Details of nesting and vectoring behaviour are given in the ARMv6-M Architecture Reference Manual.

All NVIC registers are only accessible using word transfers. Any attempt to read or write a halfword or byte individually is unpredictable.

NVIC registers are always little-endian.

Processor exception handling is described in Exceptions section.

2.4.5.1.1. SysTick timer

A 24-bit SysTick system timer, extends the functionality of both the processor and the NVIC and provides:

- A 24-bit system timer (SysTick).
- Additional configurable priority SysTick interrupt.

The SysTick timer uses a 1 μ s pulse as a clock enable. This is generated in the watchdog block as timer_tick. Accuracy of SysTick timing depends upon accuracy of this timer_tick. The SysTick timer can also run from the system clock (see [SYST_CALIB](#)).

See the ARMv6-M Architecture Reference Manual for more information.

2.4.5.1.2. Low power modes

The implementation includes a WIC. This enables the processor and NVIC to be put into a very low-power sleep mode leaving the WIC to identify and prioritize interrupts.

The processor fully implements the Wait For Interrupt (WFI), Wait For Event (WFE) and the Send Event (SEV) instructions. In addition, the processor also supports the use of SLEEPONEXIT, that causes the processor core to enter sleep mode when it returns from an exception handler to Thread mode. See the ARMv6-M Architecture Reference Manual for more information.

2.4.5.2. NVIC register summary

[Table 85](#) shows the NVIC registers. Each of these registers is 32 bits wide.

Table 85. M0+ NVIC registers

Name	Description
NVIC_IER	Interrupt Set-Enable Register.
NVIC_ICER	Interrupt Clear-Enable Register.
NVIC_ISPR	Interrupt Set-Pending Register.
NVIC_ICPR	Interrupt Clear-Pending Register.
NVIC_IPR0 - NVIC_IPR7	Interrupt Priority Registers.

Note

See the List of Registers or ARMv6-M Architecture Reference Manual for more information about the NVIC registers and their addresses, access types, and reset values.

注意

“嵌套”指中断能够被优先级更高的中断打断。“向量化”指硬件将每个中断分派至由向量表指定的独立处理程序例程。有关嵌套和向量行为的详细描述，见 ARMv6-M 架构参考手册。

所有 NVIC 寄存器仅可通过字（word）传输访问。任何单独读取或写入半字（halfword）或字节（byte）均属于不可预测操作。

NVIC 寄存器始终采用小端序格式。

处理器异常处理详见“异常”章节。

2.4.5.1.1. SysTick 计时器

24 位 SysTick 系统计时器，扩展处理器及 NVIC 功能，提供如下功能：

- 24 位系统计时器（SysTick）。
- 额外的可配置优先级 SysTick 中断。

SysTick 计时器使用 $1\mu s$ 脉冲作为时钟使能信号。此信号作为 timer_tick 在看门狗模块中生成。SysTick 定时的准确性依赖于该 timer_tick 的准确性。SysTick 定时器亦可由系统时钟驱动（参见 SYST_CALIB）。

欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

2.4.5.1.2. 低功耗模式

该实现包含 WIC，允许处理器及 NVIC 进入极低功耗休眠模式，由 WIC 负责识别与优先处理中断。

处理器完整支持 Wait For Interrupt (WFI)、Wait For Event (WFE) 及 Send Event (SEV) 指令。此外，处理器亦支持 SLEEP/NEXIT 功能，使处理器内核在异常处理程序返回线程模式时进入睡眠状态。欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

2.4.5.2. NVIC 寄存器概述

表 85 显示了 NVIC 寄存器。每个寄存器宽度均为 32 位。

表 85. M0+ NVIC 寄存器

名称	描述
NVIC_ISER	中断使能置位寄存器。
NVIC_ICER	中断使能清除寄存器。
NVIC_ISPR	中断挂起置位寄存器。
NVIC_ICPR	中断挂起清除寄存器。
NVIC_IPR0 - NVIC_IPR7	中断优先级寄存器。

注意

有关 NVIC 寄存器及其地址、访问类型及复位值的详细信息，请参阅寄存器列表或 ARMv6-M 架构参考手册。

2.4.6. MPU

2.4.6.1. About the MPU

The MPU is a component for memory protection which allows the processor to support the ARMv6 Protected Memory System Architecture model. The MPU provides full support for:

- Eight unified protection regions.
- Overlapping protection regions, with ascending region priority:
 - 7 = highest priority.
 - 0 = lowest priority.
- Access permissions.
- Exporting memory attributes to the system.

MPU mismatches and permission violations invoke the HardFault handler. See the ARMv6-M Architecture Reference Manual for more information.

You can use the MPU to:

- Enforce privilege rules.
- Separate processes.
- Manage memory attributes.

2.4.6.2. MPU register summary

[Table 86](#) shows the MPU registers. Each of these registers is 32 bits wide.

Table 86. M0+ MPU registers

Name	Description
MPU_TYPE	MPU Type Register.
MPU_CTRL	MPU Control Register.
MPU_RNR	MPU Region Number Register.
MPU_RBAR	MPU Region Base Address Register.
MPU_RASR	MPU Region Attribute and Size Register.

Note

- See the ARMv6-M Architecture Reference Manual for more information about the MPU registers and their addresses, access types, and reset values.
- The MPU supports region sizes from 256-bytes to 4Gb, with 8-sub regions per region.

2.4.7. Debug

Basic debug functionality includes processor halt, single-step, processor core register access, Reset and HardFault Vector Catch, unlimited software breakpoints, and full system memory access. See the ARMv6-M Architecture Reference Manual.

The debug features for this device are:

- A breakpoint unit supporting 4 hardware breakpoints.

2.4.6. MPU

2.4.6.1 关于MPU

MPU是实现内存保护的组件，使处理器能够支持ARMv6受保护内存系统架构模型。MPU全面支持以下功能：

- 八个统一的保护区域。
- 支持重叠的保护区域，区域优先级递增：
 - 7 = 最高优先级。
 - 0 = 最低优先级。
- 访问权限控制。
- 向系统导出内存属性。

MPU不匹配和权限违规将触发HardFault异常处理程序。欲了解更多信息，请参阅 ARMv6-M 架构参考手册。

您可以使用MPU来实现：

- 执行特权规则。
- 分离进程。
- 管理内存属性。

2.4.6.2. MPU寄存器概述

表86显示了MPU寄存器。每个寄存器宽度均为32位。

表86. M0+ MPU
寄存器

名称	描述
MPU_TYPE	MPU类型寄存器。
MPU_CTRL	MPU控制寄存器。
MPU_RNR	MPU区域编号寄存器。
MPU_RBAR	MPU区域基地址寄存器。
MPU_RASR	MPU区域属性及大小寄存器。

注意

- 有关MPU寄存器及其地址、访问类型和复位值的详细信息，请参见ARMv6-M架构参考手册。
- MPU支持256字节至4Gb的区域大小，每个区域包含8个子区域。

2.4.7. 调试

基本调试功能包括处理器停止、单步执行、处理器核心寄存器访问、复位和硬故障向量捕获、无限制软件断点及完整系统内存访问。请参见ARMv6-M架构参考手册。

本设备的调试功能包括：

- 支持4个硬件断点的断点单元。

- A watchpoint unit supporting 2 watchpoints.

2.4.8. List of Registers

The ARM Cortex-M0+ registers start at a base address of `0xe0000000` (defined as `PPB_BASE` in SDK).

Table 87. List of M0PLUS registers

Offset	Name	Info
0xe010	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register
0xe018	SYST_CVR	SysTick Current Value Register
0xe01c	SYST_CALIB	SysTick Calibration Value Register
0xe100	NVIC_ISER	Interrupt Set-Enable Register
0xe180	NVIC_ICER	Interrupt Clear-Enable Register
0xe200	NVIC_ISPR	Interrupt Set-Pending Register
0xe280	NVIC_ICPR	Interrupt Clear-Pending Register
0xe400	NVIC_IPR0	Interrupt Priority Register 0
0xe404	NVIC_IPR1	Interrupt Priority Register 1
0xe408	NVIC_IPR2	Interrupt Priority Register 2
0xe40c	NVIC_IPR3	Interrupt Priority Register 3
0xe410	NVIC_IPR4	Interrupt Priority Register 4
0xe414	NVIC_IPR5	Interrupt Priority Register 5
0xe418	NVIC_IPR6	Interrupt Priority Register 6
0xe41c	NVIC_IPR7	Interrupt Priority Register 7
0xed00	CPUID	CPUID Base Register
0xed04	ICSR	Interrupt Control and State Register
0xed08	VTOR	Vector Table Offset Register
0xed0c	AIRCR	Application Interrupt and Reset Control Register
0xed10	SCR	System Control Register
0xed14	CCR	Configuration and Control Register
0xed1c	SHPR2	System Handler Priority Register 2
0xed20	SHPR3	System Handler Priority Register 3
0xed24	SHCSR	System Handler Control and State Register
0xed90	MPU_TYPE	MPU Type Register
0xed94	MPU_CTRL	MPU Control Register
0xed98	MPU_RNR	MPU Region Number Register
0xed9c	MPU_RBAR	MPU Region Base Address Register
0xedaa	MPU_RASR	MPU Region Attribute and Size Register

M0PLUS: SYST_CSR Register

- 支持2个观察点的观察点单元。

2.4.8. 寄存器列表

ARM Cortex-M0+ 寄存器起始基址为 `0xe0000000`（在 SDK 中定义为 `PPB_BASE`）。

表 87。
M0PLUS 寄存器列表

偏移量	名称	说明
0xe010	SYST_CSR	SysTick控制与状态寄存器
0xe014	SYST_RVR	SysTick 重装载值寄存器
0xe018	SYST_CVR	SysTick 当前值寄存器
0xe01c	SYST_CALIB	SysTick 校准值寄存器
0xe100	NVIC_ISER	中断使能设置寄存器
0xe180	NVIC_ICER	中断使能清除寄存器
0xe200	NVIC_ISPR	中断挂起设置寄存器
0xe280	NVIC_ICPR	中断挂起清除寄存器
0xe400	NVIC_IPR0	中断优先级寄存器 0
0xe404	NVIC_IPR1	中断优先级寄存器1
0xe408	NVIC_IPR2	中断优先级寄存器2
0xe40c	NVIC_IPR3	中断优先级寄存器3
0xe410	NVIC_IPR4	中断优先级寄存器4
0xe414	NVIC_IPR5	中断优先级寄存器5
0xe418	NVIC_IPR6	中断优先级寄存器6
0xe41c	NVIC_IPR7	中断优先级寄存器7
0xed00	CPUID	CPUID基准寄存器
0xed04	ICSR	中断控制与状态寄存器
0xed08	VTOR	向量表偏移寄存器
0xed0c	AIRCR	应用程序中断及复位控制寄存器
0xed10	SCR	系统控制寄存器
0xed14	CCR	配置与控制寄存器
0xed1c	SHPR2	系统处理器优先级寄存器2
0xed20	SHPR3	系统处理器优先级寄存器3
0xed24	SHCSR	系统处理控制和状态寄存器
0xed90	MPU_TYPE	MPU类型寄存器
0xed94	MPU_CTRL	MPU控制寄存器
0xed98	MPU_RNR	MPU区域编号寄存器
0xed9c	MPU_RBAR	MPU 区域基址寄存器
0xedaa	MPU_RASR	MPU 区域属性与大小寄存器

M0PLUS: SYST_CSR 寄存器

Offset: 0xe010**Description**

Use the SysTick Control and Status Register to enable the SysTick features.

Table 88. SYST_CSR Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	COUNTFLAG: Returns 1 if timer counted to 0 since last time this was read. Clears on read by application or debugger.	RO	0x0
15:3	Reserved.	-	-
2	CLKSOURCE: SysTick clock source. Always reads as one if SYST_CALIB reports NOREF. Selects the SysTick timer clock source: 0 = External reference clock. 1 = Processor clock.	RW	0x0
1	TICKINT: Enables SysTick exception request: 0 = Counting down to zero does not assert the SysTick exception request. 1 = Counting down to zero asserts the SysTick exception request.	RW	0x0
0	ENABLE: Enable SysTick counter: 0 = Counter disabled. 1 = Counter enabled.	RW	0x0

M0PLUS: SYST_RVR Register**Offset:** 0xe014**Description**

Use the SysTick Reload Value Register to specify the start value to load into the current value register when the counter reaches 0. It can be any value between 0 and 0x00FFFFFF. A start value of 0 is possible, but has no effect because the SysTick interrupt and COUNTFLAG are activated when counting from 1 to 0. The reset value of this register is UNKNOWN.

To generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.

Table 89. SYST_RVR Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	RELOAD: Value to load into the SysTick Current Value Register when the counter reaches 0.	RW	0x000000

M0PLUS: SYST_CVR Register**Offset:** 0xe018**Description**

Use the SysTick Current Value Register to find the current value in the register. The reset value of this register is UNKNOWN.

Table 90. SYST_CVR Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-

偏移: 0xe010

描述

使用 SysTick 控制与状态寄存器启用 SysTick 功能。

表 88. SYST_CSR
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	COUNTFLAG ：若计时器自上次读取后计数到 0，则返回 1。 由应用程序或调试器读取时清除该标志。	只读	0x0
15:3	保留。	-	-
2	CLKSOURCE ：SysTick 时钟源。若 SYST_CALIB 报告 NOREF，则始终读取为 1。 选择 SysTick 计时器时钟源： 0 = 外部参考时钟。 1 = 处理器时钟。	读写	0x0
1	TICKINT ：启用 SysTick 异常请求： 0 = 计数到零时不发生 SysTick 异常请求。 1 = 计数到零时发生 SysTick 异常请求。	读写	0x0
0	ENABLE ：启用 SysTick 计数器： 0 = 计数器已禁用。 1 = 计数器已启用。	读写	0x0

M0PLUS: SYST_RVR 寄存器

偏移量: 0xe014

说明

使用 SysTick 重载值寄存器指定计数器计数至 0 时加载到当前值寄存器的起始值。该值可为介于 0 至 0xFFFFFFFF 之间的任意值。起始值可设为 0，但无效，因为 SysTick 中断及 COUNTFLAG 仅在从 1 计数到 0 时激活。该寄存器的复位值未知。

欲生成周期为 N 个处理器时钟周期的多次定时器，请将 RELOAD 设为 N-1。例如，若要求每 100 个时钟周期触发一次 SysTick 中断，则将 RELOAD 设为 99。

表 89. SYST_RVR
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	RELOAD ：计数器达到 0 时加载至 SysTick 当前值寄存器的数值。	读写	0x000000

M0PLUS: SYST_CVR 寄存器

偏移量: 0xe018

描述

使用 SysTick 当前值寄存器以获取寄存器中的当前值。该寄存器的复位值未知。

表 90. SYST_CVR
寄存器

位	描述	类型	复位值
31:24	保留。	-	-

Bits	Description	Type	Reset
23:0	CURRENT: Reads return the current value of the SysTick counter. This register is write-clear. Writing to it with any value clears the register to 0. Clearing this register also clears the COUNTFLAG bit of the SysTick Control and Status Register.	RW	0x000000

M0PLUS: SYST_CALIB Register

Offset: 0xe01c

Description

Use the SysTick Calibration Value Register to enable software to scale to any required speed using divide and multiply.

Table 91. SYST_CALIB Register

Bits	Description	Type	Reset
31	NOREF: If reads as 1, the Reference clock is not provided - the CLKSOURCE bit of the SysTick Control and Status register will be forced to 1 and cannot be cleared to 0.	RO	0x0
30	SKEW: If reads as 1, the calibration value for 10ms is inexact (due to clock frequency).	RO	0x0
29:24	Reserved.	-	-
23:0	TENMS: An optional Reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as 0, the calibration value is not known.	RO	0x000000

M0PLUS: NVIC_ISER Register

Offset: 0xe100

Description

Use the Interrupt Set-Enable Register to enable interrupts and determine which interrupts are currently enabled.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

Table 92. NVIC_ISER Register

Bits	Description	Type	Reset
31:0	SETENA: Interrupt set-enable bits. Write: 0 = No effect. 1 = Enable interrupt. Read: 0 = Interrupt disabled. 1 = Interrupt enabled.	RW	0x00000000

M0PLUS: NVIC_ICER Register

Offset: 0xe180

Description

Use the Interrupt Clear-Enable Registers to disable interrupts and determine which interrupts are currently enabled.

位	描述	类型	复位值
23:0	CURRENT: 读取时返回 SysTick 计数器的当前值。该寄存器为写入清零寄存器。向其写入任意值均会将寄存器清零为 0。清除此寄存器同时清除 SysTick 控制和状态寄存器中的 COUNTFLAG 位。	读写	0x000000

M0PLUS: SYST_CALIB 寄存器

偏移量: 0xe01c

描述

使用 SysTick 校准值寄存器，使软件能够通过除法和乘法缩放至任意所需速度。

表 91. SYST_CALIB 寄存器

位	描述	类型	复位值
31	NOREF: 若读取值为 1，表示未提供参考时钟——SysTick 控制和状态寄存器中的 CLKSOURCE 位将被强制为 1，且无法清零为 0。	只读	0x0
30	SKEW: 若读数为 1，则10ms 的校准值不准确（因时钟频率所致）。	只读	0x0
29:24	保留。	-	-
23:0	TENMS: 用于10ms (100Hz) 定时的可选重装载值，受系统时钟偏差误差影响。若数值为0，则表示校准值未知。	只读	0x000000

M0PLUS: NVIC_ISER 寄存器

偏移量: 0xe100

说明

使用中断置使能寄存器以启用中断，并确定当前已启用的中断。

若挂起的中断已被使能，NVIC 将根据其优先级激活该中断。若中断未被使能，断言其中断信号将使中断状态变为挂起，但无论优先级如何，NVIC 均不会激活该中断。

表 92. NVIC_ISER 寄存器

位	描述	类型	复位值
31:0	SETENA: 中断置使能位。 写入： 0 = 无效操作。 1 = 使能中断。 读取： 0 = 中断禁用。 1 = 中断使能。	读写	0x00000000

M0PLUS: NVIC_ICER 寄存器

偏移: 0xe180

描述

使用中断清除使能寄存器以禁用中断，并确定当前启用的中断。

Table 93. NVIC_ICER Register

Bits	Description	Type	Reset
31:0	CLRENA: Interrupt clear-enable bits. Write: 0 = No effect. 1 = Disable interrupt. Read: 0 = Interrupt disabled. 1 = Interrupt enabled.	RW	0x00000000

MOPLUS: NVIC_ISPR Register

Offset: 0xe200

Description

The NVIC_ISPR forces interrupts into the pending state, and shows which interrupts are pending.

Table 94. NVIC_ISPR Register

Bits	Description	Type	Reset
31:0	SETPEND: Interrupt set-pending bits. Write: 0 = No effect. 1 = Changes interrupt state to pending. Read: 0 = Interrupt is not pending. 1 = Interrupt is pending. Note: Writing 1 to the NVIC_ISPR bit corresponding to: An interrupt that is pending has no effect. A disabled interrupt sets the state of that interrupt to pending.	RW	0x00000000

MOPLUS: NVIC_ICPR Register

Offset: 0xe280

Description

Use the Interrupt Clear-Pending Register to clear pending interrupts and determine which interrupts are currently pending.

Table 95. NVIC_ICPR Register

Bits	Description	Type	Reset
31:0	CLRPEND: Interrupt clear-pending bits. Write: 0 = No effect. 1 = Removes pending state and interrupt. Read: 0 = Interrupt is not pending. 1 = Interrupt is pending.	RW	0x00000000

MOPLUS: NVIC_IPR0 Register

Offset: 0xe400

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Note: Writing 1 to an NVIC_ICPR bit does not affect the active state of the corresponding interrupt.

These registers are only word-accessible

表 93. NVIC_ICER 寄存器

位	描述	类型	复位值
31:0	CLRENA: 中断清除使能位。 写入： 0 = 无效操作。 1 = 禁用中断。 读取： 0 = 中断禁用。 1 = 中断使能。	读写	0x00000000

M0PLUS: NVIC_ISPR 寄存器

偏移: 0xe200

描述

NVIC_ISPR 强制中断进入待决状态，并显示哪些中断处于待决状态。

表 94. NVIC_ISPR 寄存器

位	描述	类型	复位值
31:0	SETPEND: 中断置待决位。 写入： 0 = 无效操作。 1 = 将中断状态更改为待决。 读取： 0 = 中断未处于待决状态。 1 = 中断处于待决状态。 注意：向 NVIC_ISPR 相关位写入 1 时： 一个已处于待决状态的中断无任何影响。 一个被禁用的中断会将该中断状态设置为待决。	读写	0x00000000

M0PLUS: NVIC_ICPR 寄存器

偏移: 0xe280

说明

使用中断清除待处理寄存器以清除待处理的中断，并确定当前哪些中断处于待处理状态。

表 95. NVIC_ICPR 寄存器

位	描述	类型	复位值
31:0	CLRPEND: 中断清除待处理位。 写入： 0 = 无效操作。 1 = 清除待处理状态及中断。 读取： 0 = 中断未处于待决状态。 1 = 中断处于待决状态。	读写	0x00000000

M0PLUS: NVIC_IPR0 寄存器

偏移: 0xe400

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

注意：向NVIC_ICPR位写入1不会影响对应中断的激活状态。

这些寄存器仅支持字访问。

Table 96. NVIC_IPR0 Register

Bits	Description	Type	Reset
31:30	IP_3: Priority of interrupt 3	RW	0x0
29:24	Reserved.	-	-
23:22	IP_2: Priority of interrupt 2	RW	0x0
21:16	Reserved.	-	-
15:14	IP_1: Priority of interrupt 1	RW	0x0
13:8	Reserved.	-	-
7:6	IP_0: Priority of interrupt 0	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR1 Register

Offset: 0xe404

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 97. NVIC_IPR1 Register

Bits	Description	Type	Reset
31:30	IP_7: Priority of interrupt 7	RW	0x0
29:24	Reserved.	-	-
23:22	IP_6: Priority of interrupt 6	RW	0x0
21:16	Reserved.	-	-
15:14	IP_5: Priority of interrupt 5	RW	0x0
13:8	Reserved.	-	-
7:6	IP_4: Priority of interrupt 4	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR2 Register

Offset: 0xe408

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 98. NVIC_IPR2 Register

Bits	Description	Type	Reset
31:30	IP_11: Priority of interrupt 11	RW	0x0
29:24	Reserved.	-	-
23:22	IP_10: Priority of interrupt 10	RW	0x0
21:16	Reserved.	-	-
15:14	IP_9: Priority of interrupt 9	RW	0x0
13:8	Reserved.	-	-

表96. NVIC_IPR0
寄存器

位	描述	类型	复位值
31:30	IP_3: 中断3的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_2: 中断2的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_1: 中断1的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_0: 中断0的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR1寄存器

偏移: 0xe404

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 97. NVIC_IPR1
寄存器

位	描述	类型	复位值
31:30	IP_7: 中断7的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_6: 中断6的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_5: 中断5的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_4: 中断4的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR2 寄存器

偏移: 0xe408

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 98. NVIC_IPR2
寄存器

位	描述	类型	复位值
31:30	IP_11: 中断11的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_10: 中断10的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_9: 中断9的优先级	读写	0x0
13:8	保留。	-	-

Bits	Description	Type	Reset
7:6	IP_8: Priority of interrupt 8	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR3 Register

Offset: 0xe40c

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 99. NVIC_IPR3 Register

Bits	Description	Type	Reset
31:30	IP_15: Priority of interrupt 15	RW	0x0
29:24	Reserved.	-	-
23:22	IP_14: Priority of interrupt 14	RW	0x0
21:16	Reserved.	-	-
15:14	IP_13: Priority of interrupt 13	RW	0x0
13:8	Reserved.	-	-
7:6	IP_12: Priority of interrupt 12	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR4 Register

Offset: 0xe410

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 100. NVIC_IPR4 Register

Bits	Description	Type	Reset
31:30	IP_19: Priority of interrupt 19	RW	0x0
29:24	Reserved.	-	-
23:22	IP_18: Priority of interrupt 18	RW	0x0
21:16	Reserved.	-	-
15:14	IP_17: Priority of interrupt 17	RW	0x0
13:8	Reserved.	-	-
7:6	IP_16: Priority of interrupt 16	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR5 Register

Offset: 0xe414

位	描述	类型	复位值
7:6	IP_8: 中断8的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR3 寄存器

偏移: 0xe40c

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 99. NVIC_IPR3
寄存器

位	描述	类型	复位值
31:30	IP_15: 中断15的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_14: 中断14的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_13: 中断13优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_12: 中断12优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR4寄存器

偏移: 0xe410

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 100. NVIC_IPR4
寄存器

位	描述	类型	复位值
31:30	IP_19: 中断19优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_18: 中断18优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_17: 中断17优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_16: 中断16优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR5寄存器

偏移: 0xe414

Description

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 101. NVIC_IPR5 Register

Bits	Description	Type	Reset
31:30	IP_23: Priority of interrupt 23	RW	0x0
29:24	Reserved.	-	-
23:22	IP_22: Priority of interrupt 22	RW	0x0
21:16	Reserved.	-	-
15:14	IP_21: Priority of interrupt 21	RW	0x0
13:8	Reserved.	-	-
7:6	IP_20: Priority of interrupt 20	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR6 Register**Offset:** 0xe418**Description**

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 102. NVIC_IPR6 Register

Bits	Description	Type	Reset
31:30	IP_27: Priority of interrupt 27	RW	0x0
29:24	Reserved.	-	-
23:22	IP_26: Priority of interrupt 26	RW	0x0
21:16	Reserved.	-	-
15:14	IP_25: Priority of interrupt 25	RW	0x0
13:8	Reserved.	-	-
7:6	IP_24: Priority of interrupt 24	RW	0x0
5:0	Reserved.	-	-

M0PLUS: NVIC_IPR7 Register**Offset:** 0xe41c**Description**

Use the Interrupt Priority Registers to assign a priority from 0 to 3 to each of the available interrupts. 0 is the highest priority, and 3 is the lowest.

Table 103. NVIC_IPR7 Register

Bits	Description	Type	Reset
31:30	IP_31: Priority of interrupt 31	RW	0x0
29:24	Reserved.	-	-
23:22	IP_30: Priority of interrupt 30	RW	0x0
21:16	Reserved.	-	-

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表 101. NVIC_IPR5
寄存器

位	描述	类型	复位值
31:30	IP_23 : 中断23优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_22 : 中断22优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_21 : 中断21优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_20 : 中断20优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR6寄存器

偏移: 0xe418

说明

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表102. NVIC_IPR6
寄存器

位	描述	类型	复位值
31:30	IP_27 : 中断27的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_26 : 中断26的优先级	读写	0x0
21:16	保留。	-	-
15:14	IP_25 : 中断25的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_24 : 中断24的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: NVIC_IPR7寄存器

偏移: 0xe41c

描述

使用中断优先级寄存器为每个可用中断分配0至3的优先级。0为最高优先级，3为最低优先级。

表103. NVIC_IPR7
寄存器

位	描述	类型	复位值
31:30	IP_31 : 中断31的优先级	读写	0x0
29:24	保留。	-	-
23:22	IP_30 : 中断30的优先级	读写	0x0
21:16	保留。	-	-

Bits	Description	Type	Reset
15:14	IP_29: Priority of interrupt 29	RW	0x0
13:8	Reserved.	-	-
7:6	IP_28: Priority of interrupt 28	RW	0x0
5:0	Reserved.	-	-

M0PLUS: CPUID Register

Offset: 0xed00

Description

Read the CPU ID Base Register to determine: the ID number of the processor core, the version number of the processor core, the implementation details of the processor core.

Table 104. CPUID Register

Bits	Description	Type	Reset
31:24	IMPLEMENTER: Implementor code: 0x41 = ARM	RO	0x41
23:20	VARIANT: Major revision number n in the rnpm revision status: 0x0 = Revision 0.	RO	0x0
19:16	ARCHITECTURE: Constant that defines the architecture of the processor: 0xC = ARMv6-M architecture.	RO	0xc
15:4	PARTNO: Number of processor within family: 0xC60 = Cortex-M0+	RO	0xc60
3:0	REVISION: Minor revision number m in the rnpm revision status: 0x1 = Patch 1.	RO	0x1

M0PLUS: ICSR Register

Offset: 0xed04

Description

Use the Interrupt Control State Register to set a pending Non-Maskable Interrupt (NMI), set or clear a pending PendSV, set or clear a pending SysTick, check for pending exceptions, check the vector number of the highest priority pended exception, check the vector number of the active exception.

Table 105. ICSR Register

Bits	Description	Type	Reset
31	NMIPENDSET: Setting this bit will activate an NMI. Since NMI is the highest priority exception, it will activate as soon as it is registered. NMI set-pending bit. Write: 0 = No effect. 1 = Changes NMI exception state to pending. Read: 0 = NMI exception is not pending. 1 = NMI exception is pending. Because NMI is the highest-priority exception, normally the processor enters the NMI exception handler as soon as it detects a write of 1 to this bit. Entering the handler then clears this bit to 0. This means a read of this bit by the NMI exception handler returns 1 only if the NMI signal is reasserted while the processor is executing that handler.	RW	0x0

位	描述	类型	复位值
15:14	IP_29 : 中断29的优先级	读写	0x0
13:8	保留。	-	-
7:6	IP_28 : 中断28的优先级	读写	0x0
5:0	保留。	-	-

M0PLUS: CPUID寄存器

偏移: 0xed00

描述

读取CPU ID基础寄存器以确定：处理器内核的ID编号、处理器内核的版本号及处理器内核的实现细节。

表104. CPUID
寄存器

位	描述	类型	复位值
31:24	IMPLEMENTER : 实现者代码: 0x41 = ARM	只读	0x41
23:20	VARIANT : rnpm修订状态中的主修订号n： 0x0 = 修订版0。	只读	0x0
19:16	ARCHITECTURE : 定义处理器架构的常量： 0xC = ARMv6-M架构。	只读	0xc
15:4	PARTNO : 系列内处理器编号: 0xC60 = Cortex-M0+	只读	0xc60
3:0	REVISION : rnpm修订状态中的次修订号m： 0x1 = 补丁1。	只读	0x1

M0PLUS: ICSR寄存器

偏移量: 0xed04

说明

使用中断控制状态寄存器设置挂起的不可屏蔽中断（NMI），设置或清除挂起的PendSV，设置或清除挂起的SysTick，检测挂起的异常，检查优先级最高的挂起异常向量号，检查当前活动异常向量号。

表105. ICSR
寄存器

位	描述	类型	复位值
31	NMIPENDSET : 设置此位将激活NMI。鉴于NMI为最高优先级异常，一经置位立即激活。 NMI 设定挂起位。 写： 0 = 无效操作。 1 = 将 NMI 异常状态更改为挂起。 读取： 0 = NMI 异常未挂起。 1 = NMI 异常已挂起。 由于NMI属于最高优先级异常，处理器通常在检测到对该位写入1时立即进入NMI 异常处理程序。进入该处理程序后，该位将被清除为0。 这意味着NMI异常处理程序读取该位时，只有在处理程序执行期间NMI信号被重新置位，才返回1。	读写	0x0

Bits	Description	Type	Reset
30:29	Reserved.	-	-
28	PENDSVSET: PendSV set-pending bit. Write: 0 = No effect. 1 = Changes PendSV exception state to pending. Read: 0 = PendSV exception is not pending. 1 = PendSV exception is pending. Writing 1 to this bit is the only way to set the PendSV exception state to pending.	RW	0x0
27	PENDSVCLR: PendSV clear-pending bit. Write: 0 = No effect. 1 = Removes the pending state from the PendSV exception.	RW	0x0
26	PENDSTSET: SysTick exception set-pending bit. Write: 0 = No effect. 1 = Changes SysTick exception state to pending. Read: 0 = SysTick exception is not pending. 1 = SysTick exception is pending.	RW	0x0
25	PENDSTCLR: SysTick exception clear-pending bit. Write: 0 = No effect. 1 = Removes the pending state from the SysTick exception. This bit is WO. On a register read its value is Unknown.	RW	0x0
24	Reserved.	-	-
23	ISRPREEMPT: The system can only access this bit when the core is halted. It indicates that a pending interrupt is to be taken in the next running cycle. If C_MASKINTS is clear in the Debug Halting Control and Status Register, the interrupt is serviced.	RO	0x0
22	ISRPENDING: External interrupt pending flag	RO	0x0
21	Reserved.	-	-
20:12	VECTPENDING: Indicates the exception number for the highest priority pending exception: 0 = no pending exceptions. Non zero = The pending state includes the effect of memory-mapped enable and mask registers. It does not include the PRIMASK special-purpose register qualifier.	RO	0x000
11:9	Reserved.	-	-
8:0	VECTACTIVE: Active exception number field. Reset clears the VECTACTIVE field.	RO	0x000

M0PLUS: VTOR Register

Offset: 0xed08

Description

The VTOR holds the vector table offset address.

位	描述	类型	复位值
30:29	保留。	-	-
28	PENDSVSET ： PendSV 挂起设置位。 写入： 0 = 无效操作。 1 = 将PendSV异常状态设为挂起。 读取： 0 = PendSV 异常未处于挂起状态。 1 = PendSV 异常处于挂起状态。 将该位写入 1 是设置 PendSV 异常状态为挂起的唯一方法。	读写	0x0
27	PENDSVCLR ： PendSV 清除挂起位。 写入： 0 = 无效操作。 1 = 从 PendSV 异常中移除挂起状态。	读写	0x0
26	PENDSTSET ： SysTick 异常置位挂起位。 写入： 0 = 无效操作。 1 = 将 SysTick 异常状态更改为挂起。 读取： 0 = SysTick 异常未处于挂起状态。 1 = SysTick 异常处于挂起状态。	读写	0x0
25	PENDSTCLR ： SysTick 异常清除挂起位。 写入： 0 = 无效操作。 1 = 移除 SysTick 异常的挂起状态。 该位为只写 (WO)。寄存器读取时，其值为未知。	读写	0x0
24	保留。	-	-
23	ISRPREEMPT ： 仅当核心停止时，系统方可访问此位。指示将在下一运行周期处理一个挂起中断。若调试停止控制与状态寄存器中的 C_MASKINTS 位清零，则中断将得到服务。	只读	0x0
22	ISR PENDING ： 外部中断挂起标志。	只读	0x0
21	保留。	-	-
20:12	VECTPENDING ： 指示最高优先级挂起异常的异常编号：0 = 无挂起异常。非零值 = 挂起状态包括内存映射的使能和屏蔽寄存器的效应。但不包括 PRIMASK 专用寄存器限定符的影响。	只读	0x000
11:9	保留。	-	-
8:0	VECTACTIVE ： 活动异常编号字段。复位将清除 VECTACTIVE 字段。	只读	0x000

M0PLUS： VTOR寄存器

偏移：0xed08

描述

VTOR寄存器保存向量表偏移地址。

Table 106. VTOR Register

Bits	Description	Type	Reset
31:8	TBLOFF: Bits [31:8] of the indicate the vector table offset address.	RW	0x000000
7:0	Reserved.	-	-

MOPLUS: AIRCR Register

Offset: 0xed0c

Description

Use the Application Interrupt and Reset Control Register to: determine data endianness, clear all active state information from debug halt mode, request a system reset.

Table 107. AIRCR Register

Bits	Description	Type	Reset
31:16	VECTKEY: Register key: Reads as Unknown On writes, write 0x05FA to VECTKEY, otherwise the write is ignored.	RW	0x0000
15	ENDIANESS: Data endianness implemented: 0 = Little-endian.	RO	0x0
14:3	Reserved.	-	-
2	SYSRESETREQ: Writing 1 to this bit causes the SYSRESETREQ signal to the outer system to be asserted to request a reset. The intention is to force a large system reset of all major components except for debug. The C_HALT bit in the DHCSR is cleared as a result of the system reset requested. The debugger does not lose contact with the device.	RW	0x0
1	VECTCLRACTIVE: Clears all active state information for fixed and configurable exceptions. This bit is self-clearing, can only be set by the DAP when the core is halted. When set: clears all active exception status of the processor, forces a return to Thread mode, forces an IPSR of 0. A debugger must re-initialize the stack.	RW	0x0
0	Reserved.	-	-

MOPLUS: SCR Register

Offset: 0xed10

Description

System Control Register. Use the System Control Register for power-management functions: signal to the system when the processor can enter a low power state, control how the processor enters and exits low power states.

Table 108. SCR Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-

表106. VTOR
寄存器

位	描述	类型	复位值
31:8	TBLOFF : 指示向量表偏移地址的[31:8]位。	读写	0x000000
7:0	保留。	-	-

M0PLUS: AIRCR寄存器

偏移: 0xed0c

描述

使用应用中断与复位控制寄存器以：确定数据字节序，清除调试暂停模式下的所有活动状态信息，发起系统复位请求。

表107. AIRCR
寄存器

位	描述	类型	复位值
31:16	VECTKEY : 寄存器密钥： 读取值为未知 写入时，须向VECTKEY写入0x05FA，否则写入操作将被忽略。	读写	0x0000
15	ENDIANESS : 实施的数据字节序： 0 = 小端序。	只读	0x0
14:3	保留。	-	-
2	SYSRESETREQ : 写1到该位将触发向外部系统发送SYSRESETREQ信号以请求复位。该操作旨在强制对除调试之外的所有主要组件进行系统复位。由于请求系统复位，DHCSR中的C_HALTI位被清除。调试器不会与设备失去连接。	读写	0x0
1	VECTCLRACTIVE : 清除所有固定和可配置异常的活动状态信息。 该位：自动清除，仅在核心停止时由DAP设置。设置时：清除处理器的所有活动异常状态，强制返回线程模式，并将IPSR强制置为0。调试器必须重新初始化堆栈。	读写	0x0
0	保留。	-	-

M0PLUS: SCR寄存器

偏移: 0xed10

描述

系统控制寄存器。用于电源管理功能：当处理器可进入低功耗状态时向系统发出信号，并控制处理器如何进入及退出低功耗状态。

表108。SCR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-

Bits	Description	Type	Reset
4	SEVONPEND: Send Event on Pending bit: 0 = Only enabled interrupts or events can wakeup the processor, disabled interrupts are excluded. 1 = Enabled events and all interrupts, including disabled interrupts, can wakeup the processor. When an event or interrupt becomes pending, the event signal wakes up the processor from WFE. If the processor is not waiting for an event, the event is registered and affects the next WFE. The processor also wakes up on execution of an SEV instruction or an external event.	RW	0x0
3	Reserved.	-	-
2	SLEEPDEEP: Controls whether the processor uses sleep or deep sleep as its low power mode: 0 = Sleep. 1 = Deep sleep.	RW	0x0
1	SLEEPONEXIT: Indicates sleep-on-exit when returning from Handler mode to Thread mode: 0 = Do not sleep when returning to Thread mode. 1 = Enter sleep, or deep sleep, on return from an ISR to Thread mode. Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application.	RW	0x0
0	Reserved.	-	-

M0PLUS: CCR Register

Offset: 0xed14

Description

The Configuration and Control Register permanently enables stack alignment and causes unaligned accesses to result in a Hard Fault.

Table 109. CCR Register

Bits	Description	Type	Reset
31:10	Reserved.	-	-
9	STKALIGN: Always reads as one, indicates 8-byte stack alignment on exception entry. On exception entry, the processor uses bit[9] of the stacked PSR to indicate the stack alignment. On return from the exception it uses this stacked bit to restore the correct stack alignment.	RO	0x0
8:4	Reserved.	-	-
3	UNALIGN_TRP: Always reads as one, indicates that all unaligned accesses generate a HardFault.	RO	0x0
2:0	Reserved.	-	-

M0PLUS: SHPR2 Register

Offset: 0xed1c

Description

System handlers are a special class of exception handler that can have their priority set to any of the priority levels.

位	描述	类型	复位值
4	<p>SEVONPEND: 挂起时发送事件位： 0 = 仅启用的中断或事件可唤醒处理器，禁用的中断除外。 1 = 启用的事件及所有中断，包括禁用的中断，均可唤醒处理器。 当事件或中断处于待处理状态时，事件信号会唤醒处于WFE状态的处理器。 如果 处理器未处于等待事件状态，该事件将被登记并影响 下一次的WFE。 处理器也会因执行SEV指令或外部事件而唤醒。</p>	读写	0x0
3	保留。	-	-
2	<p>SLEEPDEEP: 控制处理器使用睡眠或深度睡眠作为低功耗模式： 0 = 睡眠。 1 = 深度睡眠。</p>	读写	0x0
1	<p>SLEEPONEXIT: 指示从处理程序模式返回线程模式时是否进入睡眠状态： 0 = 返回线程模式时不进入睡眠。 1 = 从中断服务程序返回线程模式时进入睡眠或深度睡眠。 将此位设为1，使基于中断的应用程序避免返回空闲的主程序。</p>	读写	0x0
0	保留。	-	-

M0PLUS: CCR 寄存器

偏移: 0xed14

描述

配置与控制寄存器永久启用栈对齐，并使非对齐访问导致硬错误（Hard Fault）。

表109. CCR
寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9	<p>STKALIGN: 始终读为1，表示异常进入时8字节栈对齐。异常进入时，处理器使用堆栈中PSR的第9位指示栈对齐。异常返回时，处理器利用该堆栈位恢复正确的栈对齐。</p>	只读	0x0
8:4	保留。	-	-
3	UNALIGN_TRP : 始终读为1，表示所有非对齐访问均触发硬错误。	只读	0x0
2:0	保留。	-	-

M0PLUS: SHPR2寄存器

偏移: 0xed1c

描述

系统处理程序是特殊的异常处理程序，可将其优先级设为任一等级。

Use the System Handler Priority Register 2 to set the priority of SVCall.

Table 110. SHPR2 Register

Bits	Description	Type	Reset
31:30	PRI_11 : Priority of system handler 11, SVCall	RW	0x0
29:0	Reserved.	-	-

M0PLUS: SHPR3 Register

Offset: 0xed20

Description

System handlers are a special class of exception handler that can have their priority set to any of the priority levels. Use the System Handler Priority Register 3 to set the priority of PendSV and SysTick.

Table 111. SHPR3 Register

Bits	Description	Type	Reset
31:30	PRI_15 : Priority of system handler 15, SysTick	RW	0x0
29:24	Reserved.	-	-
23:22	PRI_14 : Priority of system handler 14, PendSV	RW	0x0
21:0	Reserved.	-	-

M0PLUS: SHCSR Register

Offset: 0xed24

Description

Use the System Handler Control and State Register to determine or clear the pending status of SVCall.

Table 112. SHCSR Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15	SVCALLPENDED : Reads as 1 if SVCall is Pending. Write 1 to set pending SVCall, write 0 to clear pending SVCall.	RW	0x0
14:0	Reserved.	-	-

M0PLUS: MPU_TYPE Register

Offset: 0xed90

Description

Read the MPU Type Register to determine if the processor implements an MPU, and how many regions the MPU supports.

Table 113. MPU_TYPE Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:16	IREGION : Instruction region. Reads as zero as ARMv6-M only supports a unified MPU.	RO	0x00
15:8	DREGION : Number of regions supported by the MPU.	RO	0x08
7:1	Reserved.	-	-

使用系统处理器优先级寄存器2（System Handler Priority Register 2）设置SVCALL的优先级。

表110. SHPR2
寄存器

位	描述	类型	复位值
31:30	PRI_11 : 系统处理器11（SVCALL）的优先级	读写	0x0
29:0	保留。	-	-

M0PLUS: SHPR3寄存器

偏移: 0xed20

描述

系统处理器是特殊的异常处理器，可将其优先级设为任一等级。

使用系统处理器优先级寄存器3设置PendSV和SysTick的优先级。

表111. SHPR3
寄存器

位	描述	类型	复位值
31:30	PRI_15 : 系统处理器15（SysTick）的优先级	读写	0x0
29:24	保留。	-	-
23:22	PRI_14 : 系统处理器14（PendSV）的优先级	读写	0x0
21:0	保留。	-	-

M0PLUS: SHCSR寄存器

偏移: 0xed24

描述

使用系统处理器控制和状态寄存器以确定或清除SVCALL的挂起状态。

表112. SHCSR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	SVCALLPENDED : 若SVCALL挂起，则读值为1。写入1以设置SVCALL挂起，写入0以清除挂起状态。	读写	0x0
14:0	保留。	-	-

M0PLUS: MPU_TYPE寄存器

偏移: 0xed90

描述

读取MPU类型寄存器以确定处理器是否实现MPU及MPU支持的区域数量。

表113. MPU_TYPE
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	IREGION : 指令区域。读取值为零，因为ARMv6-M仅支持统一MPU。	只读	0x00
15:8	DREGION : MPU支持的区域数量。	只读	0x08
7:1	保留。	-	-

Bits	Description	Type	Reset
0	SEPARATE: Indicates support for separate instruction and data address maps. Reads as 0 as ARMv6-M only supports a unified MPU.	RO	0x0

M0PLUS: MPU_CTRL Register

Offset: 0xed94

Description

Use the MPU Control Register to enable and disable the MPU, and to control whether the default memory map is enabled as a background region for privileged accesses, and whether the MPU is enabled for HardFaults and NMIs.

Table 114. MPU_CTRL Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	PRIVDEFENA: Controls whether the default memory map is enabled as a background region for privileged accesses. This bit is ignored when ENABLE is clear. 0 = If the MPU is enabled, disables use of the default memory map. Any memory access to a location not covered by any enabled region causes a fault. 1 = If the MPU is enabled, enables use of the default memory map as a background region for privileged software accesses. When enabled, the background region acts as if it is region number -1. Any region that is defined and enabled has priority over this default map.	RW	0x0
1	HFNMIENA: Controls the use of the MPU for HardFaults and NMIs. Setting this bit when ENABLE is clear results in UNPREDICTABLE behaviour. When the MPU is enabled: 0 = MPU is disabled during HardFault and NMI handlers, regardless of the value of the ENABLE bit. 1 = the MPU is enabled during HardFault and NMI handlers.	RW	0x0
0	ENABLE: Enables the MPU. If the MPU is disabled, privileged and unprivileged accesses use the default memory map. 0 = MPU disabled. 1 = MPU enabled.	RW	0x0

M0PLUS: MPU_RNR Register

Offset: 0xed98

Description

Use the MPU Region Number Register to select the region currently accessed by MPU_RBAR and MPU_RASR.

Table 115. MPU_RNR Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3:0	REGION: Indicates the MPU region referenced by the MPU_RBAR and MPU_RASR registers. The MPU supports 8 memory regions, so the permitted values of this field are 0-7.	RW	0x0

M0PLUS: MPU_RBAR Register

Offset: 0xed9c

位	描述	类型	复位值
0	SEPARATE : 指示支持独立的指令和数据地址映射。读取值为0，因为ARMv6-M仅支持统一MPU。	只读	0x0

M0PLUS: MPU_CTRL 寄存器

偏移: 0xed94

描述

使用MPU控制寄存器启用或禁用MPU，并控制是否将默认内存映射作为特权访问的背景区域启用，以及是否在HardFault和NMI中启用MPU。

表114. MPU_CTRL
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	PRIVDEFENA : 控制是否将默认内存映射作为特权软件访问的背景区域启用。当ENABLE位清零时，该位被忽略。 0 = 若启用MPU，则禁用默认内存映射。对未被任何启用区域覆盖的位置的任何内存访问均会导致故障 。 1 = 若启用MPU，则允许使用默认内存映射作为特权软件访问的背景区域。 启用时，背景区域视为区域编号 -1。任何已定义且启用的区域均优先于此默认映射。	读写	0x0
1	HFNMIENA : 控制 MPU 在 HardFault 和 NMI 中断期间的使用。若 ENABLE 位为清除状态时设置此位，将导致不可预测的行为。 当 MPU 启用时： 0 = 无论 ENABLE 位的值，MPU 在 HardFault 和 NMI 处理程序期间均被禁用。 1 = MPU 在 HardFault 和 NMI 处理程序期间启用。	读写	0x0
0	ENABLE : 启用 MPU。若 MPU 被禁用，特权和非特权访问将使用默认内存映射。 0 = MPU 禁用。 1 = MPU 启用。	读写	0x0

M0PLUS: MPU_RNR 寄存器

偏移: 0xed98

描述

使用 MPU 区域编号寄存器选择当前由 MPU_RBAR 和 MPU_RASR 访问的区域。

表 115. MPU_RNR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	REGION : 指示 MPU_RBAR 和 MPU_RASR 寄存器所引用的 MPU 区域 。 MPU 支持 8 个内存区域，因此此字段允许的值为 0-7。	读写	0x0

M0PLUS: MPU_RBAR 寄存器

偏移: 0xed9c

Description

Read the MPU Region Base Address Register to determine the base address of the region identified by MPU_RNR. Write to update the base address of said region or that of a specified region, with whose number MPU_RNR will also be updated.

Table 116. MPU_RBAR Register

Bits	Description	Type	Reset
31:8	ADDR: Base address of the region.	RW	0x000000
7:5	Reserved.	-	-
4	VALID: On writes, indicates whether the write must update the base address of the region identified by the REGION field, updating the MPU_RNR to indicate this new region. Write: 0 = MPU_RNR not changed, and the processor: Updates the base address for the region specified in the MPU_RNR. Ignores the value of the REGION field. 1 = The processor: Updates the value of the MPU_RNR to the value of the REGION field. Updates the base address for the region specified in the REGION field. Always reads as zero.	RW	0x0
3:0	REGION: On writes, specifies the number of the region whose base address to update provided VALID is set written as 1. On reads, returns bits [3:0] of MPU_RNR.	RW	0x0

M0PLUS: MPU_RASR Register**Offset:** 0xeda0**Description**

Use the MPU Region Attribute and Size Register to define the size, access behaviour and memory type of the region identified by MPU_RNR, and enable that region.

Table 117. MPU_RASR Register

Bits	Description	Type	Reset
31:16	ATTRS: The MPU Region Attribute field. Use to define the region attribute control. 28 = XN: Instruction access disable bit: 0 = Instruction fetches enabled. 1 = Instruction fetches disabled. 26:24 = AP: Access permission field 18 = S: Shareable bit 17 = C: Cacheable bit 16 = B: Bufferable bit	RW	0x0000
15:8	SRD: Subregion Disable. For regions of 256 bytes or larger, each bit of this field controls whether one of the eight equal subregions is enabled.	RW	0x00
7:6	Reserved.	-	-
5:1	SIZE: Indicates the region size. Region size in bytes = $2^{(SIZE+1)}$. The minimum permitted value is 7 (b00111) = 256Bytes	RW	0x00
0	ENABLE: Enables the region.	RW	0x0

描述

读取 MPU 区域基地址寄存器以确定由 MPU_RNR 标识区域的基地址。
写入以更新该区域或指定区域的基地址，同时 MPU_RNR 也将更新为该区域编号。

表 116. MPU_RBAR
寄存器

位	描述	类型	复位值
31:8	ADDR: 区域的基地址。	读写	0x000000
7:5	保留。	-	-
4	VALID: 写入时，指示写入是否必须更新 REGION 字段标识区域的基地址，并更新 MPU_RNR 以指示该新区域。 写入： 0 = MPU_RNR 不变，处理器： 更新 MPU_RNR 指定区域的基地址。 忽略 REGION 字段的值。 1 = 处理器： 将 MPU_RNR 更新为 REGION 字段的值。 更新 REGION 字段指定区域的基地址。 始终读取为零。	读写	0x0
3:0	REGION: 写入时指定区域编号，用以更新基地址，条件是 VALID 被写为1。读取时，返回 MPU_RNR 的第 [3:0] 位。	读写	0x0

M0PLUS: MPU_RASR 寄存器

偏移: 0xedaa0

说明

使用 MPU 区域属性及大小寄存器定义由 MPU_RNR 指定区域的大小、访问行为及内存类型，并启用该区域。

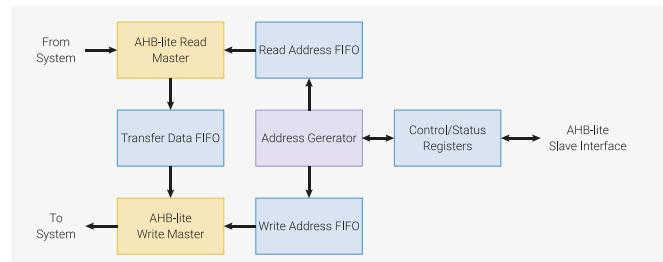
表 117. MPU_RASR
寄存器

位	描述	类型	复位值
31:16	ATTRS: MPU 区域属性字段。用于定义区域属性控制。 28 = XN：指令访问禁用位： 0 = 允许指令取指。 1 = 禁止指令取指。 26:24 = AP：访问权限字段 18 = S：共享位 17 = C：可缓存位 16 = B：缓冲位	读写	0x0000
15:8	SRD: 子区域禁用。对于大小为256字节及以上的区域，此字段的每个位控制八个等分子区域中的一个是否启用。	读写	0x00
7:6	保留。	-	-
5:1	SIZE: 表示区域大小。区域大小（字节）= $2^{(SIZE+1)}$ 。允许的最小值为7（b00111）= 256字节。	读写	0x00
0	ENABLE: 启用该区域。	读写	0x0

2.5. DMA

The RP2040 Direct Memory Access (DMA) controller has separate read and write master connections to the bus fabric, and performs bulk data transfers on a processor's behalf. This leaves processors free to attend to other tasks, or enter low-power sleep states. The data throughput of the DMA is also significantly higher than one of RP2040's processors.

Figure 12. DMA Architecture Overview.
The read master can read data from some address every clock cycle. Likewise, the write master can write to another address. The address generator produces matched pairs of read and write addresses, which the masters consume through the address FIFOs. Up to 12 transfer sequences may be in progress simultaneously, supervised by software via the control and status registers.



The DMA can perform one read access and one write access, up to 32 bits in size, every clock cycle. There are 12 independent channels, each which supervise a sequence of bus transfers, usually in one of the following scenarios:

- **Memory-to-peripheral:** a peripheral signals the DMA when it needs more data to transmit. The DMA reads data from an array in RAM or flash, and writes to the peripheral's data FIFO.
- **Peripheral-to-memory:** a peripheral signals the DMA when it has received data. The DMA reads this data from the peripheral's data FIFO, and writes it to an array in RAM.
- **Memory-to-memory:** the DMA transfers data between two buffers in RAM, as fast as possible.

Each channel has its own control and status registers (CSRs), with which software can program and monitor the channel's progress. When multiple channels are active at the same time, the DMA shares bandwidth evenly between the channels, with round-robin over all channels which are currently requesting data transfers.

The transfer size can be either 32, 16, or 8 bits. This is configured once per channel: source transfer size and destination transfer size are the same. The DMA performs standard byte lane replication on narrow writes, so byte data is available in all 4 bytes of the databus, and halfword data in both halfwords.

Channels can be combined in varied ways for more sophisticated behaviour and greater autonomy. For example, one channel can configure another, loading configuration data from a sequence of control blocks in memory, and the second can then call back to the first via the `CHAIN_TO` option, when it needs to be reconfigured.

Making the DMA more autonomous means that much less processor supervision is required: overall this allows the system to do more at once, or to dissipate less power.

2.5.1. Configuring Channels

Each channel has four control/status registers:

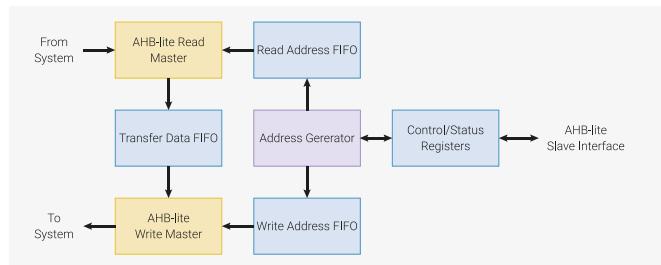
- `READ_ADDR` is a pointer to the next address to be read from
- `WRITE_ADDR` is a pointer to the next address to be written to
- `TRANS_COUNT` shows the number of transfers remaining in the current transfer sequence, and is used to program the number of transfers in the next transfer sequence (see [Section 2.5.1.2](#)).
- `CTRL` is used to configure all other aspects of the channel's behaviour, to enable/disable it, and to check for completion.

These are live registers: they update continuously as the channel progresses.

2.5. DMA

RP2040直接内存访问（DMA）控制器具备独立的读写主控接口，连接至总线结构，并代表处理器执行大容量数据传输。这使处理器可以自由处理其他任务，或进入低功耗睡眠状态。DMA的数据吞吐量显著高于RP2040的单个处理器。

图12。 DMA
架构概述。
读主控每个时钟周期均可从某地址读取数据。同样，写主控可以向另一个地址写入数据。
地址生成器产生配对的读写地址，主控通过地址FIFO使用这些地址。最多可同时进行12个传输序列，由软件通过控制与状态寄存器监督。



DMA每个时钟周期可执行一次最多32位的数据读访问和一次写访问。共有12个独立通道，每个通道负责监督一系列总线传输，通常适用于以下场景之一：

- 内存至外设：当外设需要更多数据进行传输时，会向DMA发出信号。DMA从RAM或闪存中的数组读取数据，并写入外设的数据FIFO。
- 外设至内存：当外设接收到数据时，会向DMA发出信号。DMA从外设的数据FIFO读取该数据，并写入RAM中的数组。
- 内存至内存：DMA在RAM的两个缓冲区之间尽可能快速地传输数据。

每个通道都配有独立的控制和状态寄存器（CSRs），软件可以通过它们对通道进行编程和进度监控。当多个通道同时激活时，DMA会在所有正在请求数据传输的通道间均匀分配带宽，采用轮询方式。

传输的数据大小可以是32位、16位或8位。每个通道仅需配置一次：源传输大小与目标传输大小保持一致。DMA在窄幅写入时执行标准字节通道复制，因此字节数据在数据总线的所有4个字节均可用，半字数据则在两个半字处均可用。

通道可通过多种方式组合，以实现更为复杂的行为和更高的自主性。例如，一个通道可以配置另一个通道，借助内存中一系列控制块加载配置信息；第二个通道在需要重新配置时，可通过 [CHAIN_TO](#) 选项回调第一个通道。

提高DMA的自主性意味着处理器监督需求大幅减少：总体而言，这使系统能够同时执行更多任务，或有效降低功耗。

2.5.1. 通道配置

每个通道包含四个控制/状态寄存器：

- **READ_ADDR** 指向下一次读取的地址
- **WRITE_ADDR** 指向下一次写入的地址
- **TRANS_COUNT** 显示当前传输序列中剩余的传输次数，并用于设定下一传输序列的传输次数（详见第2.5.1.2节）。
- **CTRL** 用于配置通道行为的所有其他方面，以启用或禁用该通道，并检查其完成状态。

这些为实时寄存器：它们会随通道的运行不断更新。

2.5.1.1. Read and Write Addresses

`READ_ADDR` and `WRITE_ADDR` contain the address the channel will next read from, and write to, respectively. These registers update automatically after each read/write access. They increment by 1, 2 or 4 bytes at a time, depending on the transfer size configured in `CTRL`.

Software should generally program these registers with new start addresses each time a new transfer sequence starts. If `READ_ADDR` and `WRITE_ADDR` are not reprogrammed, the DMA will use the current values as start addresses for the next transfer. For example:

- If the address does not increment (e.g. it is the address of a peripheral FIFO), and the next transfer sequence is to/from that same address, there is no need to write to the register again.
- When transferring to/from a consecutive series of buffers in memory (e.g. scattering and gathering), an address register will already have incremented to the start of the next buffer at the completion of a transfer.

By not programming all four CSRs for each transfer sequence, software can use shorter interrupt handlers, and more compact control block formats when used with channel chaining (see register aliases in [Section 2.5.2.1](#), chaining in [Section 2.5.2.2](#)).

⚠ CAUTION

`READ_ADDR` and `WRITE_ADDR` must always be aligned to the current transfer size, as specified in `CTRL.DATA_SIZE`. It is up to software to ensure the initial values are correctly aligned.

2.5.1.2. Transfer Count

Reading from `TRANS_COUNT` yields the number of transfers remaining in the current transfer sequence. This value updates continuously as the channel progresses. Writing to `TRANS_COUNT` sets the length of the *next* transfer sequence. Up to $2^{32}-1$ transfers can be performed in one sequence.

Each time the channel starts a new transfer sequence, the most recent value written to `TRANS_COUNT` is copied to the live transfer counter, which will then start to decrement again as the new transfer sequence makes progress. For debugging purposes, the last value written can be read from the `DBG_TCR` (`TRANS_COUNT` reload value) register.

If the channel is triggered multiple times without intervening writes to `TRANS_COUNT`, it performs the same number of transfers each time. For example, when chained to, one channel might load a fixed-size control block into another channel's CSRs. `TRANS_COUNT` would be programmed once by software, and then reload automatically every time.

Alternatively, `TRANS_COUNT` can be written with a new value before starting each transfer sequence. If `TRANS_COUNT` is the channel trigger (see [Section 2.5.2.1](#)), the channel will start immediately, and the value just written will be used, *not* the value currently in the reload register.

ℹ NOTE

The `TRANS_COUNT` is the number of *transfers* to be performed. The total number of bytes transferred is `TRANS_COUNT` times the size of each transfer in bytes, given by `CTRL.DATA_SIZE`.

2.5.1.3. Control/Status

The `CTRL` register has more, smaller fields than the other 3 registers, and full details of these are given in the `CTRL` register listings. Among other things, `CTRL` is used to:

- Configure the size of this channel's data transfers, via `CTRL.DATA_SIZE`. Reads and writes are the same size.
- Configure if and how `READ_ADDR` and `WRITE_ADDR` increment after each read or write, via `CTRL.INCR_WRITE`, `CTRL.INCR_READ`, `CTRL.RING_SEL`, `CTRL.RING_SIZE`. Ring transfers are available, where one of the address pointers wraps at some power-of-2 boundary.

2.5.1.1. 读写地址

`READ_ADDR`和`WRITE_ADDR`分别包含通道下一步将读取和写入的地址。这些寄存器会在每次读写访问后自动更新。根据`CTRL`中配置的传输大小，它们每次递增1、2或4字节。

软件通常应在每次新的传输序列开始时，向这些寄存器写入新的起始地址。

若`READ_ADDR`和`WRITE_ADDR`未重新编程，DMA将使用当前值作为下一次传输的起始地址。例如：

- 如果地址不递增（例如地址指向外设FIFO），且下一传输序列也是来自或写入该相同地址，则无需再次写入该寄存器。
- 在内存中传输连续缓冲区序列（例如分散和聚集）时，地址寄存器将在一次传输完成时自动递增至下一个缓冲区的起始地址。

通过不为每个传输序列编程全部四个CSR，软件可使用更短的中断处理程序，并在使用通道链时采用更紧凑的控制块格式（详见第2.5.2.1节的寄存器别名及第2.5.2.2节的链式传输）。

⚠ 注意

`READ_ADDR`和`WRITE_ADDR`必须始终与当前传输大小对齐，该传输大小由`CTRL.DATA_SIZE`指定。软件需确保初始值的正确对齐。

2.5.1.2. 传输计数

读取`TRANS_COUNT`可获知当前传输序列中剩余的传输次数。该数值会随着通道传输的进行持续更新。写入`TRANS_COUNT`将设置下一次传输序列的长度。单个序列中最多可执行 2^{13} - 1次传输。

每当通道启动新的传输序列时，最近写入`TRANS_COUNT`的值会被复制到实时传输计数器，随后该计数器将随着新传输序列的进展开始递减。出于调试目的，可从`DBG_TCR`（`TRANS_COUNT`重载值）寄存器读取最后写入的值。

如果通道在未对`TRANS_COUNT`进行写入的情况下被多次触发，则每次都会执行相同次数的传输。例如，当串联时，一个通道可能会把固定大小的控制块加载到另一个通道的CSR中。软件仅需编程一次`TRANS_COUNT`，然后每次自动重载。

或者，可以在每次启动传输序列之前向`TRANS_COUNT`写入新的值。如果`TRANS_COUNT`是通道触发器（见第2.5.2.1节），通道将立即启动，并使用刚写入的值，而非当前重载寄存器中的值。

ℹ 注意

`TRANS_COUNT`表示要执行的传输次数。传输的总字节数为`TRANS_COUNT`乘以每次传输的字节大小，该大小由`CTRL.DATA_SIZE`指定。

2.5.1.3. 控制/状态

与其他三个寄存器相比，`CTRL`寄存器包含更多且更小的字段，详细信息请参见`CTRL`寄存器列表。除其他用途外，`CTRL`用于：

- 通过`CTRL.DATA_SIZE`配置该通道数据传输的大小。读写操作的大小相同。
- 通过`CTRL.INCR_WRITE`、`CTRL.INCR_READ`、`CTRL.RING_SEL`及`CTRL.RING_SIZE`配置是否以及如何在每次读写后递增`READ_ADDR`和`WRITE_ADDR`。支持环形传输，其中一个地址指针在某个2的幂次方边界处回绕。

- Select another channel (or none) to be triggered when this channel completes, via `CTRL.CHAIN_TO`.
- Select a peripheral data request (DREQ) signal to pace this channel's transfers, via `CTRL.TREQ_SEL`.
- See when the channel is idle, via `CTRL.BUSY`.
- See if the channel has encountered a bus error, e.g. due to a faulty address being accessed, via `CTRL.AHB_ERROR`, `CTRL.READ_ERROR`, or `CTRL.WRITE_ERROR`.

2.5.2. Starting Channels

There are three ways to start a channel:

- Writing to a channel trigger register
- A chain trigger from another channel which has just completed, and has its `CHAIN_TO` field configured
- The `MULTI_CHAN_TRIGGER` register, which can start multiple channels at once

Each of these covers different use cases. For example, trigger registers are simple and efficient when configuring and starting a channel in an interrupt service routine, and `CHAIN_TO` allows one channel to callback to another channel, which can then reconfigure the first channel.

i NOTE

Triggering a channel which is already running has no effect.

2.5.2.1. Aliases and Triggers

Table 118. Control register aliases. Each channel has four control/status registers. Each register can be accessed at multiple different addresses. In each naturally-aligned group of four, all four registers appear, in different orders.

Offset	+0x0	+0x4	+0x8	+0xC (Trigger)
0x00 (Alias 0)	READ_ADDR	WRITE_ADDR	TRANS_COUNT	CTRL_TRIG
0x10 (Alias 1)	CTRL	READ_ADDR	WRITE_ADDR	TRANS_COUNT_TRIG
0x20 (Alias 2)	CTRL	TRANS_COUNT	READ_ADDR	WRITE_ADDR_TRIG
0x30 (Alias 3)	CTRL	WRITE_ADDR	TRANS_COUNT	READ_ADD_TRIG

The four CSRs are aliased multiple times in memory. Each alias – of which there are four – exposes the same four physical registers, but in a different order. The final register in each alias (at offset `+0xC`, highlighted) is a trigger register. Writing to the trigger register starts the channel.

Often, only alias 0 is used, and aliases 1-3 can be ignored. The channel is configured *and* started by writing `READ_ADDR`, `WRITE_ADDR`, `TRANS_COUNT` and finally `CTRL`. Since `CTRL` is the trigger register in alias 0, this starts the channel.

The other aliases allow more compact control block lists when using one channel to configure another, and more efficient reconfiguration and launch in interrupt handlers:

- Each CSR is a trigger register in one of the aliases:
 - When gathering fixed-size buffers into a peripheral, the DMA channel can be configured and launched by writing only `READ_ADDR_TRIG`.
 - When scattering from a peripheral to fixed-size buffers, the channel can be configured and launched by writing only `WRITE_ADDR_TRIG`.
- Useful combinations of registers appear as naturally-aligned tuples which contain a trigger register. In conjunction with channel chaining and address wrapping, these implement compressed control block formats, e.g.:
 - (`WRITE_ADDR`, `TRANS_COUNT_TRIG`) for peripheral scatter operations

- 通过`CTRL.CHAIN_TO`选择另一通道（或无通道）以在本通道完成后触发。
- 通过`CTRL.TREQ_SEL`选择一个外设数据请求（DREQ）信号以控制该通道的传输节奏。
- 通过`CTRL.BUSY`查看通道是否空闲。
- 通过`CTRL.AHB_ERROR`、`CTRL.READ_ERROR`或`CTRL.WRITE_ERROR`查看通道是否发生总线错误，例如访问了错误地址。

2.5.2. 启动通道

启动通道有三种方式：

- 向通道触发寄存器写入数据
- 从刚完成的另一个通道触发链式启动，且其`CHAIN_TO`字段已配置
- 使用`MULTI_CHAN_TRIGGER`寄存器，可同时启动多个通道

每种方式适用于不同的使用场景。例如，触发寄存器在中断服务例程中配置并启动通道时表现出简单且高效的特点，而`CHAIN_TO`允许一个通道回调另一个通道，后者随后可重新配置第一个通道。

① 注意

触发已运行中的通道无任何效果。

2.5.2.1. 别名与触发器

表118。控制寄存器别名每个通道具有四个控制/状态寄存器。每个寄存器可以通过多个不同地址访问。在每组自然对齐的四个寄存器中，所有四个寄存器都会出现，但顺序不同。

偏移量	+0x0	+0x4	+0x8	+0xC (触发)
0x00 (别名0)	READ_ADDR	WRITE_ADDR	TRANS_COUNT	CTRL_TRIG
0x10 (别名1)	CTRL	READ_ADDR	WRITE_ADDR	TRANS_COUNT_TRIG
0x20 (别名2)	CTRL	TRANS_COUNT	READ_ADDR	WRITE_ADDR_TRIG
0x30 (别名3)	CTRL	WRITE_ADDR	TRANS_COUNT	READ_ADD_TRIG

这四个控制/状态寄存器在内存中被多重别名映射。每个别名一共计四个一一映射相同的四个物理寄存器，但顺序不同。每个别名中最后一个寄存器（偏移+0xC，已高亮）为触发寄存器。

向触发寄存器写入数据将启动该通道。

通常仅使用别名0，别名1至3可忽略。通过写入`READ_ADDR`、`WRITE_ADDR`、`TRANS_COUNT`，最后写入`CTRL`来配置并启动通道。由于`CTRL`是别名0中的触发寄存器，因此此操作启动通道。

其他别名允许在使用一个通道配置另一个时，实现更紧凑的控制块列表，以及在中断处理程序中更高效的重新配置与启动：

- 每个CSR均为某别名中的触发寄存器：
 - 在将固定大小缓冲区聚集到外设时，可仅通过写入`READ_ADDR_TRIG`配置并启动DMA通道。
 - 在外设分散到固定大小缓冲区时，可仅通过写入`WRITE_ADDR_TRIG`配置并启动通道。
- 寄存器的有效组合呈现为自然对齐的元组，其中包含一个触发寄存器。结合通道链和地址环绕，可实现压缩控制块格式，如：
 - (`WRITE_ADDR`, `TRANS_COUNT_TRIG`) 用于外设散射操作

- (TRANS_COUNT, READ_ADDR_TRIG) for peripheral gather operations, or calculating CRCs on a list of buffers
- (READ_ADDR, WRITE_ADDR_TRIG) for manipulating fixed-size buffers in memory

Trigger registers do not start the channel if:

- The channel is disabled via CTRL.EN. (If the trigger is CTRL, the just-written value of EN is used, *not* the value currently in the CTRL register.)
- The channel is already running
- The value 0 is written to the trigger register. (This is useful for ending control block chains. See null triggers, [Section 2.5.2.3](#))

2.5.2.2. Chaining

When a channel completes, it can name a different channel to immediately be triggered. This can be used as a callback for the second channel to reconfigure and restart the first.

This feature is configured through the CHAIN_TO field in the channel CTRL register. This 4-bit value selects a channel that will start when this one finishes. A channel can not chain to itself. Setting CHAIN_TO to a channel's own index means no chaining will take place.

Chain triggers behave the same as triggers from other sources, such as trigger registers. For example, they cause TRANS_COUNT to reload, and they are ignored if the targeted channel is already running.

One application for CHAIN_TO is for a channel to request reconfiguration by another channel, from a sequence of control blocks in memory. Channel A is configured to perform a wrapped transfer from memory to channel B's control registers (including a trigger register), and channel B is configured to chain back to channel A when it completes each transfer sequence. This is shown more explicitly in the DMA control blocks example ([Section 2.5.6.2](#)).

Use of the register aliases ([Section 2.5.2.1](#)) enables compact formats for DMA control blocks: as little as one word in some cases.

Another use of chaining is a "ping-pong" configuration, where two channels each trigger one another. The processor can respond to the channel completion interrupts, and reconfigure each channel after it completes; however, the chained channel, which has already been configured, starts immediately. In other words, channel configuration and channel operation are pipelined. Performance can improve dramatically where many short transfer sequences are required.

The [Section 2.5.6](#) goes into more detail on the possibilities of chain triggers, in the real world.

2.5.2.3. Null Triggers and Chain Interrupts

As mentioned in [Section 2.5.2.1](#), writing all-zeroes to a trigger register does *not* start the channel. This is called a null trigger, and it has two purposes:

- Cause a halt at the end of an array of control blocks, by appending an all-zeroes block
- Reduce the number of interrupts generated when control blocks are used

By default, a channel will generate an interrupt each time it finishes a transfer sequence, unless that channel's IRQ is masked in INTE0 or INTE1. The rate of interrupts can be excessive, particularly as processor attention is generally not required while a sequence of control blocks are in progress; however, processor attention *is* required at the end of a chain.

The channel CTRL register has a field called IRQ QUIET. Its default value is 0. When this set to 1, channels generate an interrupt when they receive a null trigger, and at no other time. The interrupt is generated by the channel which receives the trigger.

- (TRANS_COUNT, READ_ADDR_TRIG) 用于外设聚合操作，或对缓冲区列表计算 CRC
- (READ_ADDR, WRITE_ADDR_TRIG) 用于操作内存中的固定大小缓冲区

触发寄存器在以下情况下不会启动通道：

- 通道通过 CTRL.EN 被禁用。（若触发源为 CTRL，使用刚写入的 EN 值，而非 CTRL 寄存器中当前的值。）
- 通道已在运行中。
- 触发寄存器写入值为 0。（此功能用于结束控制块链。详见第 2.5.2.3 节“空触发”）

2.5.2.2. 链接

当通道完成时，可以指定另一个通道立即触发。此功能可用作第二通道的回调，用于重新配置并重启第一个通道。

该功能通过通道 CTRL 寄存器中的 CHAIN_TO 字段进行配置。该 4 位数值选择一个通道，当本通道结束时，将启动所选通道。通道不可链至自身。将 CHAIN_TO 设置为通道自身的索引表示不会发生链式连接。

链式触发的行为与来自其他源（例如触发寄存器）的触发相同。例如，它们会导致 TRANS_COUNT 重新加载，且若目标通道已运行则会被忽略。

CHAIN_TO 的一个应用是，通道请求由另一通道从内存中的控制块序列进行重新配置。通道 A 配置为从内存执行环绕传输至通道 B 的控制寄存器（包括触发寄存器），通道 B 配置为在完成每个传输序列后链回通道 A。此情形在 DMA 控制块示例（第 2.5.6.2 节）中有更明确的说明。

使用寄存器别名（第 2.5.2.1 节）可实现 DMA 控制块的紧凑格式，在某些情况下仅需一个字。

链式使用的另一种形式是“乒乓”配置，其中两个通道相互触发。处理器可以响应通道完成的中断，并在通道完成后重新配置该通道；然而，已配置完毕的链式通道会立即开始。换言之，通道配置与通道操作实现流水线处理。在需要大量短传输序列的情况下，性能可显著提升。

第 2.5.6 节详细说明了链式触发在实际应用中的可能性。

2.5.2.3. 空触发与链式中断

如第 2.5.2.1 节所述，向触发寄存器写入全零值不会启动通道。此称为空触发，具有以下两个目的：

- 通过添加全零控制块，实现控制块数组末尾的停止
- 减少使用控制块时生成的中断数量

默认情况下，每当通道完成传输序列时，都会产生中断，除非该通道的 IRQ 在 INTE0 或 INTE1 中被屏蔽。中断的频率可能过高，尤其在一系列控制块执行期间，处理器通常无需关注；但在链的末端，处理器关注是必要的。

通道 CTRL 寄存器包含一个名为 IRQ_QUIET 的字段，其默认值为 0。当该字段设置为 1 时，通道仅在接收到空触发信号时触发中断，其他情况下不触发中断。中断由接收到触发信号的通道产生。

2.5.3. Data Request (DREQ)

Peripherals produce or consume data at their own pace. If the DMA simply transferred data as fast as possible, loss or corruption of data would ensue. DREQs are a communication channel between peripherals and the DMA, which enables the DMA to pace transfers according to the needs of the peripheral.

The `CTRL.TREQ_SEL` (transfer request) field selects an external DREQ. It can also be used to select one of the internal pacing timers, or select no TREQ at all (the transfer proceeds as fast as possible), e.g. for memory-to-memory transfers.

2.5.3.1. System DREQ Table

There is a global assignment of DREQ numbers to peripheral DREQ channels.

Table 119. DREQs

DREQ	DREQ Channel	DREQ	DREQ Channel	DREQ	DREQ Channel	DREQ	DREQ Channel
0	DREQ_PI00_TX0	10	DREQ_PI01_TX2	20	DREQ_UART0_RX	30	DREQ_PWM_WRAP6
1	DREQ_PI00_TX1	11	DREQ_PI01_RX3	21	DREQ_UART0_TX	31	DREQ_PWM_WRAP7
2	DREQ_PI00_RX0	12	DREQ_PI01_RX0	22	DREQ_UART1_RX	32	DREQ_I2C0_RX
3	DREQ_PI00_RX1	13	DREQ_PI01_RX1	23	DREQ_UART1_TX	33	DREQ_I2C0_TX
4	DREQ_PI00_RX2	14	DREQ_PI01_RX2	24	DREQ_PWM_WRAP0	34	DREQ_I2C1_RX
5	DREQ_PI00_RX3	15	DREQ_PI01_RX3	25	DREQ_PWM_WRAP1	35	DREQ_I2C1_TX
6	DREQ_SPI0_RX	16	DREQ_SPI0_TX	26	DREQ_PWM_WRAP2	36	DREQ_ADC
7	DREQ_SPI0_RX	17	DREQ_SPI1_RX	27	DREQ_PWM_WRAP3	37	DREQ_XIP_STREAM
8	DREQ_SPI1_RX	18	DREQ_SPI1_TX	28	DREQ_PWM_WRAP4	38	DREQ_XIP_SSITX
9	DREQ_SPI1_RX	19	DREQ_SPI1_TX	29	DREQ_PWM_WRAP5	39	DREQ_XIP_SSIRX

2.5.3.2. Credit-based DREQ Scheme

The RP2040 DMA is designed for systems where:

- The area and power cost of large peripheral data FIFOs is prohibitive
- The bandwidth demands of individual peripherals may be high, e.g. >50% bus injection rate for short periods
- Bus latency is low, but multiple masters may be competing for bus access

In addition, the DMA's transfer FIFOs and dual-master structure permit multiple accesses to the same peripheral to be in flight at once, to improve gross throughput. Choice of DREQ mechanism is therefore critical:

- The traditional "turn on the tap" method can cause overflow if multiple writes are backed up in the TDF. Some systems solve this by overprovisioning peripheral FIFOs and setting the DREQ threshold below the full level, but this wastes precious area and power
- The ARM-style single and burst handshake does not permit additional requests to be registered while the current request is being served. This limits performance when FIFOs are very shallow.

The RP2040 DMA uses a credit-based DREQ mechanism. For each peripheral, the DMA attempts to keep as many transfers in flight as the peripheral has capacity for. This enables full bus throughput (1 word per clock) through an 8-deep peripheral FIFO with no possibility of overflow or underflow, in the absence of fabric latency or contention.

For each channel, the DMA maintains a counter. Each 1-clock pulse on the `dreq` signal will increment this counter (saturating). When nonzero, the channel requests a transfer from the DMA's internal arbiter, and the counter is decremented when the transfer is issued to the address FIFOs. At this point the transfer is in flight, but has not yet necessarily completed.

2.5.3. 数据请求 (DREQ)

外设以自身速率产生或消耗数据。若DMA仅按最大速度传输数据，将导致数据丢失或损坏。DREQ是外设与DMA之间的通信通道，使DMA能根据外设需求调整传输节奏。

`CTRL.TREQ_SEL`（传输请求）字段用于选择外部DREQ，也可选择内部节奏定时器之一，或选择不使用TREQ（传输以最大速度进行），例如内存到内存传输。

2.5.3.1. 系统 DREQ 表

DREQ 编号被全局分配至外设 DREQ 通道。

表 119. DREQs

DREQ	DREQ 通道	DREQ	DREQ 通道	DREQ	DREQ 通道	DREQ	DREQ 通道
0	DREQ_PI00_TX0	10	DREQ_PI01_TX2	20	DREQ_UART0_TX	30	DREQ_PWM_WRAP6
1	DREQ_PI00_RX1	11	DREQ_PI01_RX3	21	DREQ_UART0_RX	31	DREQ_PWM_WRAP7
2	DREQ_PI00_RX2	12	DREQ_PI01_RX0	22	DREQ_UART1_TX	32	DREQ_I2C0_RX
3	DREQ_PI00_RX3	13	DREQ_PI01_RX1	23	DREQ_UART1_RX	33	DREQ_I2C0_RX
4	DREQ_PI00_RX0	14	DREQ_PI01_RX2	24	DREQ_PWM_WRAP0	34	DREQ_I2C1_RX
5	DREQ_PI00_RX1	15	DREQ_PI01_RX3	25	DREQ_PWM_WRAP1	35	DREQ_I2C1_RX
6	DREQ_PI00_RX2	16	DREQ_SPI0_TX	26	DREQ_PWM_WRAP2	36	DREQ_ADC
7	DREQ_PI00_RX3	17	DREQ_SPI0_RX	27	DREQ_PWM_WRAP3	37	DREQ_XIP_STREAM
8	DREQ_PI01_TX0	18	DREQ_SPI1_TX	28	DREQ_PWM_WRAP4	38	DREQ_XIP_SSITX
9	DREQ_PI01_RX1	19	DREQ_SPI1_RX	29	DREQ_PWM_WRAP5	39	DREQ_XIP_SSIRX

2.5.3.2. 基于信用的 DREQ 方案

RP2040 DMA 设计适用于以下系统：

- 大型外设数据 FIFO 在面积和功耗方面的成本过高
- 个别外设的带宽需求可能较高，例如短时总线注入率超过 50%
- 总线延迟较低，但多个主控可能竞争总线访问权限

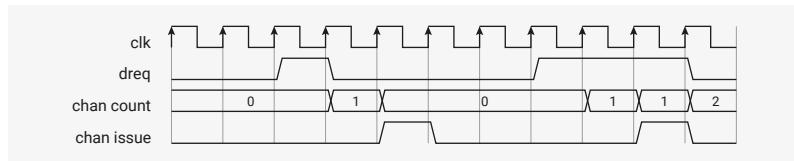
此外，DMA 的传输 FIFO 及双主控结构允许同一外设同时存在多个访问请求，从而提升整体吞吐量。因此，DREQ 机制的选择至关重要：

- 传统的“开启水龙头”方式若多个写操作在 TDF 中积压，可能导致溢出。一些系统通过为外设 FIFO 过度配置容量并将 DREQ 阈值设置在满容量以下来解决此问题，但这将浪费宝贵的面积和功耗资源。
- ARM风格的单次和突发握手协议在当前请求处理期间不允许注册额外请求。当FIFO较浅时，此机制会限制性能表现。

RP2040 DMA采用基于信用的DREQ机制。对于每个外设，DMA尝试维持尽可能多的传输同时进行，数量取决于外设的处理能力。此举实现了通过具有8级深度外设FIFO的全总线吞吐量（每时钟周期1字），在不存在布线延迟或资源竞争的情况下，避免FIFO发生溢出或欠载。

DMA为每个通道维护一个计数器。每个 `dreq`信号的1时钟脉冲会使该计数器递增（饱和计数）。计数器非零时，通道向DMA内部仲裁器请求传输；当传输下发到地址FIFO时计数器递减。此阶段传输处于进行中，但尚未完成。

Figure 13. DREQ counting



The effect is to upper bound the number of in-flight transfers based on the amount of room or data available in the peripheral FIFO. In the steady state, this gives maximum throughput, but can't underflow or overflow.

One caveat is that the user *must not* access a FIFO which is currently being serviced by the DMA. This causes the channel and peripheral to become desynchronised, and can cause corruption or loss of data.

Another caveat is that multiple channels should not be connected to the same DREQ.

2.5.4. Interrupts

Each channel can generate interrupts; these can be masked on a per-channel basis using the `INTE0` or `INTE1` registers. There are two circumstances where a channel raises an interrupt request:

- On the completion of each transfer sequence, if `CTRL.IRQ_QUIET` is disabled
- On receiving a null trigger, if `CTRL.IRQ_QUIET` is enabled

The masked interrupt status is visible in the `INTS` registers; there is one bit for each channel. Interrupts are cleared by writing a bit mask to `INTS`. One idiom for acknowledging interrupts is to read `INTS` and then write the same value back, so only enabled interrupts are cleared.

The RP2040 DMA provides two system IRQs, with independent masking and status registers (e.g. `INTE0`, `INTE1`). Any combination of channel interrupt requests can be routed to either system IRQ. For example:

- Some channels can be given a higher priority in the system interrupt controller, if they have particularly tight timing requirements
- In multiprocessor systems, different channel interrupts can be routed independently to different cores

For debugging purposes, the `INTF` registers can force either IRQ to be asserted.

2.5.5. Additional Features

2.5.5.1. Pacing Timers

These allow transfer of data roughly once every n `clk_sys` clocks instead of using external peripheral DREQ to trigger transfers. A fractional (X/Y) divider is used, and will generate a maximum of 1 request per `clk_sys` cycle.

There are 4 timers available in RP2040. Each DMA is able to select any of these in `CTRL.TREQ_SEL`.

2.5.5.2. CRC Calculation

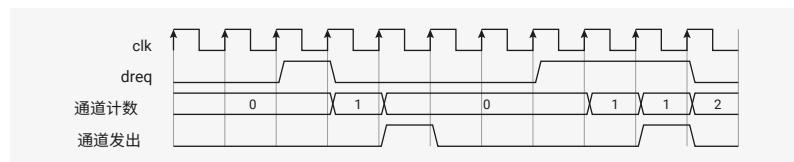
The DMA can watch data from a given channel passing through the data FIFO, and calculate checksums based on this data. This is a purely passive affair: the data is not altered by this hardware, only observed.

The feature is controlled via the `SNIFF_CTRL` and `SNIFF_DATA` registers, and can be enabled/disabled per DMA transfer via the `CTRL.SNIF_EN` field.

As this hardware cannot place backpressure on the FIFO, it must keep up with the DMA's maximum transfer rate of 32 bits per clock.

The supported checksums are:

图13. DREQ
计数



其作用是在外设FIFO中可用的空间或数据量基础上，对在途传输数量设定上限。在稳态下，这能实现最大吞吐量，且不会发生溢出或欠载。

一项注意事项是，用户不得访问当前正由DMA服务的FIFO。这将导致通道与外设不同步，可能引发数据损坏或丢失。

另一项注意事项是，不应将多个通道连接至同一DREQ。

2.5.4. 中断

每个通道均可产生中断；这些中断可通过 `INTE0` 或 `INTE1` 寄存器按通道掩码。

通道在以下两种情况下会触发中断请求：

- 每个传输序列完成时，且 `CTRL.IRQ QUIET` 被禁用时
- 接收到空触发信号时，且 `CTRL.IRQ QUIET` 被启用时

被掩码的中断状态可在INTS寄存器中查看；该寄存器为每个通道对应一个位。通过向INTS写入位掩码以清除中断。确认中断的一种惯用做法是先读取INTS，再将相同的值写回，如此仅会清除已启用的中断。

RP2040 DMA 提供两个系统 IRQ，具有独立的屏蔽和状态寄存器（例如 `INTE0` 和 `INTE1`）。任意组合的通道中断请求均可路由至任一系统 IRQ。例如：

- 若某些通道具有特别严格的时序要求，则可在系统中断控制器中赋予其更高优先级。
- 在多处理器系统中，不同通道的中断请求可独立路由至不同核心。

出于调试目的，INTF寄存器可强制触发任一IRQ的断言。

2.5.5. 附加功能

2.5.5.1. 节奏计时器

该计时器允许大致每隔 n `clk_sys` 时钟周期传输一次数据，无需通过外部外设的 DREQ 触发传输。采用分数 (X/Y) 分频器设计，每个 `clk_sys` 时钟周期最多可产生一次请求。

RP2040 中提供 4 个定时器。每个 DMA 可在 `CTRL.TREQ_SEL` 字段中选择其中任意一个。

2.5.5.2. CRC 计算

DMA 可监视指定通道经数据 FIFO 传递的数据，并基于此数据计算校验和。此过程完全被动：硬件不会修改数据，仅进行监测。

该功能由 `SNIFF_CTRL` 和 `SNIFF_DATA` 寄存器控制，且可通过 `CTRL.SNIFF_EN` 字段在每次 DMA 传输时启用或禁用。

由于此硬件无法对 FIFO 施加回压，故必须满足 DMA 最大 32 位/时钟的传输速率要求。

支持的校验和类型如下：

- CRC-32, MSB-first and LSB-first
- CRC-16-CCITT, MSB-first and LSB-first
- Simple summation (add to 32-bit accumulator)
- Even parity

The result register is both readable and writable, so that the initial seed value can be set.

Bit/byte manipulations are available on the result which may aid specific use cases:

- Bit inversion
- Bit reversal
- Byte swap

These manipulations do not affect the CRC calculation, just how the data is presented in the result register.

2.5.5.3. Channel Abort

It is possible for a channel to get into an irrecoverable state: e.g. if commanded to transfer more data than a peripheral will ever request, it will never complete. Clearing the `CTRL.EN` bit merely pauses the channel, and does not solve the problem. This should not occur under normal circumstances, but it is important that there is a mechanism to recover without simply hard-resetting the entire DMA block.

The `CHAN_ABORT` register forces channels to complete early. There is one bit for each channel, and writing a 1 terminates that channel. This clears the transfer counter and forces the channel into an inactive state.

CAUTION

Due to RP2040-E13, aborting a DMA channel that is making progress (i.e. not stalled on an inactive DREQ) may cause a completion IRQ to assert. The channel interrupt enable should be cleared before performing the abort, and the interrupt should be checked and cleared after the abort.

At the time an abort is triggered, a channel may have bus transfers currently in flight between the read and write master, and these transfers cannot be revoked. The `CTRL.BUSY` flag stays high until these transfers complete, and the channel reaches a safe state, which generally takes only a few cycles. The channel must not be restarted until its `CTRL.BUSY` flag deasserts. Starting a new sequence of transfers whilst transfers from an old sequence are still in flight can lead to unpredictable behaviour.

2.5.5.4. Debug

Debug registers are available for each DMA channel to show the dreq counter `DBG_CTDREQ` and next transfer count `DBG_TCR`. These can also be used to reset a DMA channel if required.

2.5.6. Example Use Cases

2.5.6.1. Using Interrupts to Reconfigure a Channel

When a channel finishes a block of transfers, it becomes available for making more transfers. Software detects that the channel is no longer busy, and reconfigures and restarts the channel. One approach is to poll the `CTRL_BUSY` bit until the channel is done, but this loses one of the key advantages of the DMA, namely that it does *not* have to operate in lockstep with a processor. By setting the correct bit in `INTE0` or `INTE1`, we can instruct the DMA to raise one of its two interrupt request lines when a given channel completes. Rather than repeatedly asking if a channel is done, we are told.

- CRC-32，最高有效位优先及最低有效位优先
- CRC-16-CCITT，最高有效位优先及最低有效位优先
- 简单求和（累计至32位累加器）
- 偶数奇偶校验

结果寄存器支持读写，允许设置初始种子值。

结果支持位/字节操作，可能有助于特定用例：

- 位反转
- 位逆转
- 字节交换

这些操作不会影响CRC计算，仅改变结果寄存器中数据的呈现方式。

2.5.5.3. 通道中止

通道可能进入不可恢复状态：例如，如果指令传输的数据量超出外设请求的范围，传输将永远无法完成。清除 `CTRL.EN` 位仅会暂停通道，不能解决该问题。此情况在正常操作中不应发生，但必须具备无需硬重置整个DMA模块即可恢复的机制。

`CHAN_ABORT` 寄存器可强制通道提前终止。寄存器中每个位对应一条通道，写入1则终止相应通道。此操作会清除传输计数器，并将通道置于非活动状态。

⚠ 注意

由于RP2040-E13问题，终止正在传输数据（即未因DREQ非活动而阻塞）的DMA通道，可能会触发完成中断（IRQ）。执行中止操作前，应清除通道中断使能位；中止操作后，应检查并清除中断。

触发中止时，通道可能存在读写主设备间正在进行的总线传输，且这些传输不可撤销。`CTRL.BUSY` 标志将保持高电平，直到所有传输完成且通道进入安全状态，通常仅需数个时钟周期。通道必须等 `CTRL.BUSY` 标志复位后，方可重新启动。在旧序列传输尚未完成时启动新序列的传输，可能导致不可预测行为。

2.5.5.4. 调试

每个DMA通道配备调试寄存器，用于显示dreq计数器 `BG_CTDREQ` 及下一个传输计数 `BG_TCR`。如有需要，这些寄存器也可用于复位DMA通道。

2.5.6. 示例用例

2.5.6.1. 使用中断重新配置通道

通道完成一组传输后，即可用于执行后续更多传输。软件检测到通道不再忙碌，随后重新配置并重启该通道。一种方法是轮询 `CTRL_BUSY` 位，直到通道完成，但这会失去DMA的一个关键优势，即其无需与处理器同步操作。通过设置 `INTE0` 或 `INTE1` 中的相应位，我们可以指示DMA在某个通道完成时激活其两条中断请求线之一。我们无需反复查询通道是否完成，而是直接收到通知。

NOTE

Having two system interrupt lines allows different channel completion interrupts to be routed to different cores, or to preempt one another on the same core if one channel is more time-critical.

When the interrupt is asserted, the processor can be configured to drop whatever it is doing and call a user-specified handler function. The handler can reconfigure and restart the channel. When the handler exits, the processor returns to the interrupted code running in the foreground.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c Lines 35 - 52

```

35 void dma_handler() {
36     static int pwm_level = 0;
37     static uint32_t wavetable[N_PWM_LEVELS];
38     static bool first_run = true;
39     // Entry number `i` has `i` one bits and `(32 - i)` zero bits.
40     if (first_run) {
41         first_run = false;
42         for (int i = 0; i < N_PWM_LEVELS; ++i)
43             wavetable[i] = ~(~0u << i);
44     }
45
46     // Clear the interrupt request.
47     dma_hw->ints0 = 1u << dma_chan;
48     // Give the channel a new wave table entry to read from, and re-trigger it
49     dma_channel_set_read_addr(dma_chan, &wavetable[pwm_level], true);
50
51     pwm_level = (pwm_level + 1) % N_PWM_LEVELS;
52 }
```

In many cases, most of the configuration can be done the first time the channel is started, and only addresses and transfer lengths need reprogramming in the DMA handler.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c Lines 54 - 94

```

54 int main() {
55 #ifndef PICO_DEFAULT_LED_PIN
56 #warning dma/channel_irq example requires a board with a regular LED
57 #else
58     // Set up a PIO state machine to serialise our bits
59     uint offset = pio_add_program(pio0, &pio_serialiser_program);
60     pio_serialiser_program_init(pio0, 0, offset, PICO_DEFAULT_LED_PIN, PIO_SERIAL_CLKDIV);
61
62     // Configure a channel to write the same word (32 bits) repeatedly to PIO0
63     // SM0's TX FIFO, paced by the data request signal from that peripheral.
64     dma_chan = dma_claim_unused_channel(true);
65     dma_channel_config c = dma_channel_get_default_config(dma_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, false);
68     channel_config_set_dreq(&c, DREQ_PI00_TX0);
69
70     dma_channel_configure(
71         dma_chan,
72         &c,
73         &pio0_hw->txf[0], // Write address (only need to set this once)
74         NULL,              // Don't provide a read address yet
75         PWM_REPEAT_COUNT, // Write the same value many times, then halt and interrupt
76         false              // Don't start yet
77     );
78 }
```

注意

使用两条系统中断线，允许将不同通道的完成中断路由到不同核心，或在同一核心上根据通道的时间紧迫性实现中断抢占。

当中断被触发时，处理器可以配置为暂停当前操作，调用用户指定的处理程序函数。处理程序可以重新配置并重启该通道。处理程序结束后，处理器恢复执行被中断的前台代码。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c 第35至52行

```

35 void dma_handler() {
36     static int pwm_level = 0;
37     static uint32_t wavetable[N_PWM_LEVELS];
38     static bool first_run = true;
39     // 条目编号 `i` 含有 `i` 个1位及 `(32 - i)` 个0位。
40     if (first_run) {
41         first_run = false;
42         for (int i = 0; i < N_PWM_LEVELS; ++i)
43             wavetable[i] = ~(~0u << i);
44     }
45
46     // 清除中断请求。
47     dma_hw->ints0 = 1u << dma_chan;
48     // 为该通道分配新的波形表条目以供读取，并重新触发其操作
49     dma_channel_set_read_addr(dma_chan, &wavetable[pwm_level], true);
50
51     pwm_level = (pwm_level + 1) % N_PWM_LEVELS;
52 }
```

在多数情况下，大部分配置可在通道首次启动时完成，仅需在DMA处理程序中重新编程地址及传输长度。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/channel_irq/channel_irq.c 第54至94行

```

54 int main() {
55 #ifndef PICO_DEFAULT_LED_PIN
56 // warning dma/channel_irq 示例要求开发板配备常规LED
57 #else
58     // 设置PIO状态机以序列化位流
59     uint offset = pio_add_program(pio0, &pio_serialiser_program);
60     pio_serialiser_program_init(pio0, 0, offset, PICO_DEFAULT_LED_PIN, PIO_SERIAL_CLKDIV);
61
62     // 配置通道以反复向PIO0写入相同的32位数据
63     // SM0 的 TX FIFO，由该外设的数据请求信号控制节奏。
64     dma_chan = dma_claim_unused_channel(true);
65     dma_channel_config c = dma_channel_get_default_config(dma_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, false);
68     channel_config_set_dreq(&c, DREQ_PI00_TX0);
69
70     dma_channel_configure(
71         dma_chan,
72         &c,
73         &pio0_hw->txf[0], // 写入地址（仅需设置一次）
74         NULL, // 暂不提供读取地址
75         PWM_REPEAT_COUNT, // 多次写入相同值，随后暂停并触发中断
76         false // 暂不启动
77     );
78 }
```

```

79 // Tell the DMA to raise IRQ line 0 when the channel finishes a block
80 dma_channel_set_irq0_enabled(dma_chan, true);
81
82 // Configure the processor to run dma_handler() when DMA IRQ 0 is asserted
83 irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
84 irq_set_enabled(DMA_IRQ_0, true);
85
86 // Manually call the handler once, to trigger the first transfer
87 dma_handler();
88
89 // Everything else from this point is interrupt-driven. The processor has
90 // time to sit and think about its early retirement -- maybe open a bakery?
91 while (true)
92     tight_loop_contents();
93 #endif
94 }

```

One disadvantage of this technique is that we don't start to reconfigure the channel until some time after the channel makes its last transfer. If there is heavy interrupt activity on the processor, this may be quite a long time, and therefore quite a large gap in transfers, which is problematic if we need to sustain a high data throughput.

This is solved by using two channels, with their `CHAIN_TO` fields crossed over, so that channel A triggers channel B when it completes, and vice versa. At any point in time, one of the channels is transferring data, and the other is either already configured to start the next transfer immediately when the current one finishes, or it is in the process of being reconfigured. When channel A completes, it immediately starts the cued-up transfer on channel B. At the same time, the interrupt is fired, and the handler reconfigures channel A so that it is ready for when channel B completes.

2.5.6.2. DMA Control Blocks

Frequently, multiple smaller buffers must be gathered together and sent to the same peripheral. To address this use case, the RP2040 DMA can execute a long and complex sequence of transfers without processor control. One channel repeatedly reconfigures a second channel, and the second channel restarts the first each time it completes block of transfers.

Because the first DMA channel is transferring data directly from memory to the second channel's control registers, the format of the control blocks in memory must match those registers. The last register written to, each time, will be one of the trigger registers ([Section 2.5.2.1](#)) which will start the second channel on its programmed block of transfers. The register aliases ([Section 2.5.2.1](#)) give some flexibility for the block layout, and more importantly allow some registers to be omitted from the blocks, so they occupy less memory and can be loaded more quickly.

This example shows how multiple buffers can be gathered and transferred to the UART, by reprogramming `TRANS_COUNT` and `READ_ADDR_TRIG`:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/dma/control_blocks/control_blocks.c

```

1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 // Use two DMA channels to make a programmed sequence of data transfers to the
8 // UART (a data gather operation). One channel is responsible for transferring
9 // the actual data, the other repeatedly reprograms that channel.
10
11 #include <stdio.h>
12 #include "pico/stlolib.h"
13 #include "hardware/dma.h"
14 #include "hardware/structs/uart.h"

```

```

79 // 指示 DMA 在通道完成数据块时触发 IRQ 0 线
80 dma_channel_set_irq0_enabled(dma_chan, true);
81
82 // 配置处理器在 DMA IRQ 0 触发时执行 dma_handler()
83 irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
84 irq_set_enabled(DMA_IRQ_0, true);
85
86 // 手动调用处理程序一次，以触发首次传输
87 dma_handler();
88
89 // 此后所有操作均由中断驱动。处理器有
90 // 时间思考其提前退休的计划——或许开一家面包店？
91 while (true)
92     tight_loop_contents();
93 #endif
94 }

```

此技术的一个缺点在于，必须等到通道完成最后一次传输后，才能开始重新配置通道。若处理器承受频繁的中断活动，该延迟可能相当长，导致传输间断较大，这对于需要维持高数据吞吐量的场景而言存在问题。

该问题通过使用两个通道及其 `CHAIN_TO` 字段交叉配置得以解决，即通道A完成后触发通道B，反之亦然。在任何时刻，其中一个通道正在传输数据，另一个通道要么已配置为在当前传输结束后立即启动下一次传输，要么正处于重新配置过程。当通道A完成传输时，会立即启动通道B上排队的传输。与此同时，中断被触发，处理程序会重新配置通道A，以便在通道B完成时做好准备。

2.5.6.2. DMA 控制块

通常，需要将多个较小缓冲区汇总并发送至同一外设。为满足此类需求，RP2040 DMA 可实现无需处理器干预的长且复杂的传输序列。一个通道反复重新配置第二个通道，第二个通道每完成一块传输后则重新启动第一个通道。

由于第一个DMA通道直接从内存向第二个通道的控制寄存器传输数据，内存中控制块的格式必须与这些寄存器保持一致。每次写入的最后一个寄存器，将是触发寄存器之一（第2.5.2.1节），它将启动第二通道执行其已编程的传输块。寄存器别名（第2.5.2.1节）为块布局提供了灵活性，更重要的是允许从块中省略某些寄存器，从而减少内存占用并提高加载速度。

此示例展示了如何通过重新编程 `TRANS_COUNT` 和 `READ_ADDR_TRIGGER` 来收集多个缓冲区并传输至UART：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/dma/control_blocks/control_blocks.c

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 // 使用两个DMA通道构成一个编程的数据传输序列，传输至
8 // UART (数据收集操作)。一个通道负责传输
9 // 实际数据，另一个通道则不断重新编程该通道。
10
11 #include <stdio.h>
12 #include "pico/stlolib.h"
13 #include "hardware/dma.h"
14 #include "hardware/structs/uart.h"

```

```

15
16 // These buffers will be DMA'd to the UART, one after the other.
17
18 const char word0[] = "Transferring ";
19 const char word1[] = "one ";
20 const char word2[] = "word ";
21 const char word3[] = "at ";
22 const char word4[] = "a ";
23 const char word5[] = "time.\n";
24
25 // Note the order of the fields here: it's important that the length is before
26 // the read address, because the control channel is going to write to the last
27 // two registers in alias 3 on the data channel:
28 //          +0x0      +0x4      +0x8      +0xC (Trigger)
29 // Alias 0: READ_ADDR  WRITE_ADDR  TRANS_COUNT  CTRL
30 // Alias 1: CTRL       READ_ADDR   WRITE_ADDR   TRANS_COUNT
31 // Alias 2: CTRL       TRANS_COUNT READ_ADDR    WRITE_ADDR
32 // Alias 3: CTRL       WRITE_ADDR   TRANS_COUNT  READ_ADDR
33 //
34 // This will program the transfer count and read address of the data channel,
35 // and trigger it. Once the data channel completes, it will restart the
36 // control channel (via CHAIN_TO) to load the next two words into its control
37 // registers.
38
39 const struct {uint32_t len; const char *data;} control_blocks[] = {
40     {count_of(word0) - 1, word0}, // Skip null terminator
41     {count_of(word1) - 1, word1},
42     {count_of(word2) - 1, word2},
43     {count_of(word3) - 1, word3},
44     {count_of(word4) - 1, word4},
45     {count_of(word5) - 1, word5},
46     {0, NULL}                  // Null trigger to end chain.
47 };
48
49 int main() {
50 #ifndef uart_default
51 #warning dma/control_blocks example requires a UART
52 #else
53     stdio_init_all();
54     puts("DMA control block example:");
55
56     // ctrl_chan loads control blocks into data_chan, which executes them.
57     int ctrl_chan = dma_claim_unused_channel(true);
58     int data_chan = dma_claim_unused_channel(true);
59
60     // The control channel transfers two words into the data channel's control
61     // registers, then halts. The write address wraps on a two-word
62     // (eight-byte) boundary, so that the control channel writes the same two
63     // registers when it is next triggered.
64
65     dma_channel_config c = dma_channel_get_default_config(ctrl_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, true);
68     channel_config_set_write_increment(&c, true);
69     channel_config_set_ring(&c, true, 3); // 1 << 3 byte boundary on write ptr
70
71     dma_channel_configure(
72         ctrl_chan,
73         &c,
74         &dma_hw->ch[data_chan].al3_transfer_count, // Initial write address
75         &control_blocks[0],                         // Initial read address
76         2,                                         // Halt after each control block
77         false                                      // Don't start yet
78     );

```

```

15
16 // 这些缓冲区将依次通过DMA传输至UART。
17
18 const char word0[] = "Transferring ";
19 const char word1[] = "one ";
20 const char word2[] = "word ";
21 const char word3[] = "at ";
22 const char word4[] = "a ";
23 const char word5[] = "time.\n";
24
25 // 注意字段的顺序：长度字段必须位于读取地址之前，
26 // 因为控制通道将向数据通道别名3中最后两个寄存器写入：
27 // two registers in alias 3 on the data channel:
28 //          +0x0      +0x4      +0x8      +0xC (Trigger)
29 // 别名 0: READ_ADDR  WRITE_ADDR TRANS_COUNT CTRL
30 // 别名 1: CTRL      READ_ADDR   WRITE_ADDR  TRANS_COUNT
31 // 别名 2: CTRL      TRANS_COUNT READ_ADDR   WRITE_ADDR
32 // 别名 3: CTRL      WRITE_ADDR   TRANS_COUNT READ_ADDR
33 //
34 // 这将设置数据通道的传输计数和读取地址，
35 // 并触发该通道。数据通道完成后，将重新启动
36 // 控制通道（通过 CHAIN_TO），以加载接下来的两个字到其控制
37 // 寄存器中。
38
39 const struct {uint32_t len; const char *data;} control_blocks[] = {
40     {count_of(word0) - 1, word0}, // 跳过空终止符
41     {count_of(word1) - 1, word1},
42     {count_of(word2) - 1, word2},
43     {count_of(word3) - 1, word3},
44     {count_of(word4) - 1, word4},
45     {count_of(word5) - 1, word5},
46     {0, NULL}                  // 用于结束链的空触发器。
47 };
48
49 int main() {
50 #ifndef uart_default
51 #warning dma/control_blocks 示例需要UART
52 #else
53     stdio_init_all();
54     puts("DMA 控制块示例: ");
55
56     // ctrl_chan 将控制块加载到 data_chan，后者负责执行它们。
57     int ctrl_chan = dma_claim_unused_channel(true);
58     int data_chan = dma_claim_unused_channel(true);
59
60     // 控制通道将两个字传输到数据通道的控制
61     // 寄存器，然后停止。写入地址在两个字的
62     // (八字节) 边界处循环，因此控制通道会写入相同的两个字
63     // 下次触发时的寄存器数量。
64
65     dma_channel_config c = dma_channel_get_default_config(ctrl_chan);
66     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
67     channel_config_set_read_increment(&c, true);
68     channel_config_set_write_increment(&c, true);
69     channel_config_set_ring(&c, true, 3); // 写指针的 1 << 3 字节边界
70
71     dma_channel_configure(
72         ctrl_chan,
73         &c,
74         &dma_hw->ch[data_chan].a13_transfer_count, // 初始写地址
75         &control_blocks[0],                          // 初始读地址
76         2,                                         // 每个控制块后暂停
77         false                                      // 暂不启动
78     );

```

```

79
80 // The data channel is set up to write to the UART FIFO (paced by the
81 // UART's TX data request signal) and then chain to the control channel
82 // once it completes. The control channel programs a new read address and
83 // data length, and retriggers the data channel.
84
85 c = dma_channel_get_default_config(data_chan);
86 channel_config_set_transfer_data_size(&c, DMA_SIZE_8);
87 channel_config_set_dreq(&c, uart_get_dreq(uart_default, true));
88 // Trigger ctrl_chan when data_chan completes
89 channel_config_set_chain_to(&c, ctrl_chan);
90 // Raise the IRQ flag when 0 is written to a trigger register (end of chain):
91 channel_config_set_irq_quiet(&c, true);
92
93 dma_channel_configure(
94     data_chan,
95     &c,
96     &uart_get_hw(uart_default)->dr,
97     NULL,           // Initial read address and transfer count are unimportant;
98     0,             // The control channel will reprogram them each time.
99     false          // Don't start yet.
100 );
101
102 // Everything is ready to go. Tell the control channel to load the first
103 // control block. Everything is automatic from here.
104 dma_start_channel_mask(1u << ctrl_chan);
105
106 // The data channel will assert its IRQ flag when it gets a null trigger,
107 // indicating the end of the control block list. We're just going to wait
108 // for the IRQ flag instead of setting up an interrupt handler.
109 while (!(dma_hw->intr & 1u << data_chan))
110     tight_loop_contents();
111 dma_hw->ints0 = 1u << data_chan;
112
113 puts("DMA finished.");
114 #endif
115 }

```

2.5.7. List of Registers

The DMA registers start at a base address of **0x50000000** (defined as **DMA_BASE** in SDK).

Table 120. List of DMA registers

Offset	Name	Info
0x000	CH0_READ_ADDR	DMA Channel 0 Read Address pointer
0x004	CH0_WRITE_ADDR	DMA Channel 0 Write Address pointer
0x008	CH0_TRANS_COUNT	DMA Channel 0 Transfer Count
0x00c	CH0_CTRL_TRIG	DMA Channel 0 Control and Status
0x010	CH0_AL1_CTRL	Alias for channel 0 CTRL register
0x014	CH0_AL1_READ_ADDR	Alias for channel 0 READ_ADDR register
0x018	CH0_AL1_WRITE_ADDR	Alias for channel 0 WRITE_ADDR register
0x01c	CH0_AL1_TRANS_COUNT_TRIG	Alias for channel 0 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x020	CH0_AL2_CTRL	Alias for channel 0 CTRL register

```

79
80 // 数据通道配置为写入 UART FIFO (由 UART 的 TX 数据请求信号控制)
81 // 完成后再链接至控制通道
82 // once it completes. 控制通道设置新的读取地址及
83 // 数据长度，并重新触发数据通道。
84
85 c = dma_channel_get_default_config(data_chan);
86 channel_config_set_transfer_data_size(&c, DMA_SIZE_8);
87 channel_config_set_dreq(&c, uart_get_dreq(uart_default, true));
88 // 当 data_chan 完成时，触发 ctrl_chan
89 channel_config_set_chain_to(&c, ctrl_chan);
90 // 向触发寄存器写入 0 (链结束) 时，提升 IRQ 标志：
91 channel_config_set_irq_quiet(&c, true);
92
93 dma_channel_configure(
94     data_chan,
95     &c,
96     &uart_get_hw(uart_default)->dr,
97     NULL,           // 初始读取地址和传输计数无关紧要;
98     0,              // 控制通道将每次重新编程它们。
99     false           // 暂不启动。
100 );
101
102 // 一切准备就绪，通知控制通道加载首个
103 // 控制块。此后操作全部自动完成。
104 dma_start_channel_mask(1u << ctrl_chan);
105
106 // 当数据通道收到空触发时，将断言其 IRQ 标志，
107 // 表示控制块列表结束。我们将等待 108 // IRQ 标志，而不设置中断处理程序。
108
109 while (!(dma_hw->intr & 1u << data_chan))
110     tight_loop_contents();
111 dma_hw->ints0 = 1u << data_chan;
112
113 puts("DMA 完成。");
114 #endif
115 }

```

2.5.7. 寄存器列表

DMA 寄存器起始于基地址 **0x50000000**(在 SDK 中定义为 DMA_BASE)。

表 120。DMA 寄存器列表

偏移量	名称	说明
0x000	CH0_READ_ADDR	DMA 通道 0 读地址指针
0x004	CH0_WRITE_ADDR	DMA 通道 0 写地址指针
0x008	CH0_TRANS_COUNT	DMA 通道 0 传输计数
0x00c	CH0_CTRL_TRIG	DMA 通道 0 控制与状态寄存器
0x010	CH0_AL1_CTRL	通道 0 CTRL 寄存器的别名
0x014	CH0_AL1_READ_ADDR	通道 0 READ_ADDR 寄存器的别名
0x018	CH0_AL1_WRITE_ADDR	通道 0 WRITE_ADDR 寄存器的别名
0x01c	CH0_AL1_TRANS_COUNT_TRIG	通道 0 TRANS_COUNT 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x020	CH0_AL2_CTRL	通道 0 CTRL 寄存器的别名

Offset	Name	Info
0x024	CH0_AL2_TRANS_COUNT	Alias for channel 0 TRANS_COUNT register
0x028	CH0_AL2_READ_ADDR	Alias for channel 0 READ_ADDR register
0x02c	CH0_AL2_WRITE_ADDR_TRIG	Alias for channel 0 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x030	CH0_AL3_CTRL	Alias for channel 0 CTRL register
0x034	CH0_AL3_WRITE_ADDR	Alias for channel 0 WRITE_ADDR register
0x038	CH0_AL3_TRANS_COUNT	Alias for channel 0 TRANS_COUNT register
0x03c	CH0_AL3_READ_ADDR_TRIG	Alias for channel 0 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x040	CH1_READ_ADDR	DMA Channel 1 Read Address pointer
0x044	CH1_WRITE_ADDR	DMA Channel 1 Write Address pointer
0x048	CH1_TRANS_COUNT	DMA Channel 1 Transfer Count
0x04c	CH1_CTRL_TRIG	DMA Channel 1 Control and Status
0x050	CH1_AL1_CTRL	Alias for channel 1 CTRL register
0x054	CH1_AL1_READ_ADDR	Alias for channel 1 READ_ADDR register
0x058	CH1_AL1_WRITE_ADDR	Alias for channel 1 WRITE_ADDR register
0x05c	CH1_AL1_TRANS_COUNT_TRIG	Alias for channel 1 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x060	CH1_AL2_CTRL	Alias for channel 1 CTRL register
0x064	CH1_AL2_TRANS_COUNT	Alias for channel 1 TRANS_COUNT register
0x068	CH1_AL2_READ_ADDR	Alias for channel 1 READ_ADDR register
0x06c	CH1_AL2_WRITE_ADDR_TRIG	Alias for channel 1 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x070	CH1_AL3_CTRL	Alias for channel 1 CTRL register
0x074	CH1_AL3_WRITE_ADDR	Alias for channel 1 WRITE_ADDR register
0x078	CH1_AL3_TRANS_COUNT	Alias for channel 1 TRANS_COUNT register
0x07c	CH1_AL3_READ_ADDR_TRIG	Alias for channel 1 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x080	CH2_READ_ADDR	DMA Channel 2 Read Address pointer
0x084	CH2_WRITE_ADDR	DMA Channel 2 Write Address pointer
0x088	CH2_TRANS_COUNT	DMA Channel 2 Transfer Count
0x08c	CH2_CTRL_TRIG	DMA Channel 2 Control and Status
0x090	CH2_AL1_CTRL	Alias for channel 2 CTRL register
0x094	CH2_AL1_READ_ADDR	Alias for channel 2 READ_ADDR register

偏移量	名称	说明
0x024	CH0_AL2_TRANS_COUNT	通道 0 TRANS_COUNT 寄存器的别名
0x028	CH0_AL2_READ_ADDR	通道 0 READ_ADDR 寄存器的别名
0x02c	CH0_AL2_WRITE_ADDR_TRIG	通道 0 WRITE_ADDR 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x030	CH0_AL3_CTRL	通道 0 CTRL 寄存器的别名
0x034	CH0_AL3_WRITE_ADDR	通道 0 WRITE_ADDR 寄存器的别名
0x038	CH0_AL3_TRANS_COUNT	通道 0 TRANS_COUNT 寄存器的别名
0x03c	CH0_AL3_READ_ADDR_TRIG	通道 0 READ_ADDR 寄存器的别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x040	CH1_READ_ADDR	DMA 通道 1 读地址指针
0x044	CH1_WRITE_ADDR	DMA 通道 1 写地址指针
0x048	CH1_TRANS_COUNT	DMA 通道 1 传输计数
0x04c	CH1_CTRL_TRIG	DMA 通道 1 控制与状态寄存器
0x050	CH1_AL1_CTRL	通道 1 CTRL 寄存器别名
0x054	CH1_AL1_READ_ADDR	通道 1 READ_ADDR 寄存器别名
0x058	CH1_AL1_WRITE_ADDR	通道 1 WRITE_ADDR 寄存器别名
0x05c	CH1_AL1_TRANS_COUNT_TRIG	通道 1 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x060	CH1_AL2_CTRL	通道 1 CTRL 寄存器别名
0x064	CH1_AL2_TRANS_COUNT	通道 1 TRANS_COUNT 寄存器别名
0x068	CH1_AL2_READ_ADDR	通道 1 READ_ADDR 寄存器别名
0x06c	CH1_AL2_WRITE_ADDR_TRIG	通道 1 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x070	CH1_AL3_CTRL	通道 1 CTRL 寄存器别名
0x074	CH1_AL3_WRITE_ADDR	通道 1 WRITE_ADDR 寄存器别名
0x078	CH1_AL3_TRANS_COUNT	通道 1 TRANS_COUNT 寄存器别名
0x07c	CH1_AL3_READ_ADDR_TRIG	通道 1 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x080	CH2_READ_ADDR	DMA 通道 2 读地址指针
0x084	CH2_WRITE_ADDR	DMA 通道 2 写地址指针
0x088	CH2_TRANS_COUNT	DMA 通道 2 传输计数
0x08c	CH2_CTRL_TRIG	DMA 通道 2 控制与状态寄存器
0x090	CH2_AL1_CTRL	通道 2 CTRL 寄存器别名
0x094	CH2_AL1_READ_ADDR	通道 2 READ_ADDR 寄存器别名

Offset	Name	Info
0x098	CH2_AL1_WRITE_ADDR	Alias for channel 2 WRITE_ADDR register
0x09c	CH2_AL1_TRANS_COUNT_TRIG	Alias for channel 2 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x0a0	CH2_AL2_CTRL	Alias for channel 2 CTRL register
0x0a4	CH2_AL2_TRANS_COUNT	Alias for channel 2 TRANS_COUNT register
0x0a8	CH2_AL2_READ_ADDR	Alias for channel 2 READ_ADDR register
0x0ac	CH2_AL2_WRITE_ADDR_TRIG	Alias for channel 2 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x0b0	CH2_AL3_CTRL	Alias for channel 2 CTRL register
0x0b4	CH2_AL3_WRITE_ADDR	Alias for channel 2 WRITE_ADDR register
0x0b8	CH2_AL3_TRANS_COUNT	Alias for channel 2 TRANS_COUNT register
0x0bc	CH2_AL3_READ_ADDR_TRIG	Alias for channel 2 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x0c0	CH3_READ_ADDR	DMA Channel 3 Read Address pointer
0x0c4	CH3_WRITE_ADDR	DMA Channel 3 Write Address pointer
0x0c8	CH3_TRANS_COUNT	DMA Channel 3 Transfer Count
0xcc	CH3_CTRL_TRIG	DMA Channel 3 Control and Status
0xd0	CH3_AL1_CTRL	Alias for channel 3 CTRL register
0xd4	CH3_AL1_READ_ADDR	Alias for channel 3 READ_ADDR register
0xd8	CH3_AL1_WRITE_ADDR	Alias for channel 3 WRITE_ADDR register
0xdc	CH3_AL1_TRANS_COUNT_TRIG	Alias for channel 3 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0xe0	CH3_AL2_CTRL	Alias for channel 3 CTRL register
0xe4	CH3_AL2_TRANS_COUNT	Alias for channel 3 TRANS_COUNT register
0xe8	CH3_AL2_READ_ADDR	Alias for channel 3 READ_ADDR register
0xec	CH3_AL2_WRITE_ADDR_TRIG	Alias for channel 3 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0xf0	CH3_AL3_CTRL	Alias for channel 3 CTRL register
0xf4	CH3_AL3_WRITE_ADDR	Alias for channel 3 WRITE_ADDR register
0xf8	CH3_AL3_TRANS_COUNT	Alias for channel 3 TRANS_COUNT register
0xfc	CH3_AL3_READ_ADDR_TRIG	Alias for channel 3 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x100	CH4_READ_ADDR	DMA Channel 4 Read Address pointer
0x104	CH4_WRITE_ADDR	DMA Channel 4 Write Address pointer

偏移量	名称	说明
0x098	CH2_AL1_WRITE_ADDR	通道 2 WRITE_ADDR 寄存器别名
0x09c	CH2_AL1_TRANS_COUNT_TRIG	通道 2 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0a0	CH2_AL2_CTRL	通道 2 CTRL 寄存器别名
0x0a4	CH2_AL2_TRANS_COUNT	通道 2 的 TRANS_COUNT 寄存器别名
0x0a8	CH2_AL2_READ_ADDR	通道 2 READ_ADDR 寄存器别名
0x0ac	CH2_AL2_WRITE_ADDR_TRIG	通道 2 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0b0	CH2_AL3_CTRL	通道 2 CTRL 寄存器别名
0x0b4	CH2_AL3_WRITE_ADDR	通道 2 WRITE_ADDR 寄存器别名
0x0b8	CH2_AL3_TRANS_COUNT	通道 2 的 TRANS_COUNT 寄存器别名
0x0bc	CH2_AL3_READ_ADDR_TRIG	通道 2 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x0c0	CH3_READ_ADDR	DMA 通道 3 读取地址指针
0x0c4	CH3_WRITE_ADDR	DMA 通道 3 写入地址指针
0x0c8	CH3_TRANS_COUNT	DMA 通道 3 传输计数
0xcc	CH3_CTRL_TRIG	DMA 通道 3 控制与状态寄存器
0xd0	CH3_AL1_CTRL	通道 3 的 CTRL 寄存器别名
0xd4	CH3_AL1_READ_ADDR	通道 3 的 READ_ADDR 寄存器别名
0xd8	CH3_AL1_WRITE_ADDR	通道 3 的 WRITE_ADDR 寄存器别名
0xdc	CH3_AL1_TRANS_COUNT_TRIG	通道 3 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0xe0	CH3_AL2_CTRL	通道 3 的 CTRL 寄存器别名
0xe4	CH3_AL2_TRANS_COUNT	通道 3 的 TRANS_COUNT 寄存器别名
0xe8	CH3_AL2_READ_ADDR	通道 3 的 READ_ADDR 寄存器别名
0xec	CH3_AL2_WRITE_ADDR_TRIG	通道 3 的 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0xf0	CH3_AL3_CTRL	通道 3 的 CTRL 寄存器别名
0xf4	CH3_AL3_WRITE_ADDR	通道 3 的 WRITE_ADDR 寄存器别名
0xf8	CH3_AL3_TRANS_COUNT	通道 3 的 TRANS_COUNT 寄存器别名
0xfc	CH3_AL3_READ_ADDR_TRIG	通道 3 的 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x100	CH4_READ_ADDR	DMA 通道 4 读取地址指针
0x104	CH4_WRITE_ADDR	DMA 通道 4 写入地址指针

Offset	Name	Info
0x108	CH4_TRANS_COUNT	DMA Channel 4 Transfer Count
0x10c	CH4_CTRL_TRIG	DMA Channel 4 Control and Status
0x110	CH4_AL1_CTRL	Alias for channel 4 CTRL register
0x114	CH4_AL1_READ_ADDR	Alias for channel 4 READ_ADDR register
0x118	CH4_AL1_WRITE_ADDR	Alias for channel 4 WRITE_ADDR register
0x11c	CH4_AL1_TRANS_COUNT_TRIG	Alias for channel 4 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x120	CH4_AL2_CTRL	Alias for channel 4 CTRL register
0x124	CH4_AL2_TRANS_COUNT	Alias for channel 4 TRANS_COUNT register
0x128	CH4_AL2_READ_ADDR	Alias for channel 4 READ_ADDR register
0x12c	CH4_AL2_WRITE_ADDR_TRIG	Alias for channel 4 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x130	CH4_AL3_CTRL	Alias for channel 4 CTRL register
0x134	CH4_AL3_WRITE_ADDR	Alias for channel 4 WRITE_ADDR register
0x138	CH4_AL3_TRANS_COUNT	Alias for channel 4 TRANS_COUNT register
0x13c	CH4_AL3_READ_ADDR_TRIG	Alias for channel 4 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x140	CH5_READ_ADDR	DMA Channel 5 Read Address pointer
0x144	CH5_WRITE_ADDR	DMA Channel 5 Write Address pointer
0x148	CH5_TRANS_COUNT	DMA Channel 5 Transfer Count
0x14c	CH5_CTRL_TRIG	DMA Channel 5 Control and Status
0x150	CH5_AL1_CTRL	Alias for channel 5 CTRL register
0x154	CH5_AL1_READ_ADDR	Alias for channel 5 READ_ADDR register
0x158	CH5_AL1_WRITE_ADDR	Alias for channel 5 WRITE_ADDR register
0x15c	CH5_AL1_TRANS_COUNT_TRIG	Alias for channel 5 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x160	CH5_AL2_CTRL	Alias for channel 5 CTRL register
0x164	CH5_AL2_TRANS_COUNT	Alias for channel 5 TRANS_COUNT register
0x168	CH5_AL2_READ_ADDR	Alias for channel 5 READ_ADDR register
0x16c	CH5_AL2_WRITE_ADDR_TRIG	Alias for channel 5 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x170	CH5_AL3_CTRL	Alias for channel 5 CTRL register
0x174	CH5_AL3_WRITE_ADDR	Alias for channel 5 WRITE_ADDR register
0x178	CH5_AL3_TRANS_COUNT	Alias for channel 5 TRANS_COUNT register

偏移量	名称	说明
0x108	CH4_TRANS_COUNT	DMA 通道 4 传输计数
0x10c	CH4_CTRL_TRIG	DMA 通道 4 控制与状态寄存器
0x110	CH4_AL1_CTRL	通道 4 的 CTRL 寄存器别名
0x114	CH4_AL1_READ_ADDR	通道 4 的 READ_ADDR 寄存器别名
0x118	CH4_AL1_WRITE_ADDR	通道 4 的 WRITE_ADDR 寄存器别名
0x11c	CH4_AL1_TRANS_COUNT_TRIG	通道 4 的 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x120	CH4_AL2_CTRL	通道 4 的 CTRL 寄存器别名
0x124	CH4_AL2_TRANS_COUNT	通道 4 的 TRANS_COUNT 寄存器别名
0x128	CH4_AL2_READ_ADDR	通道 4 的 READ_ADDR 寄存器别名
0x12c	CH4_AL2_WRITE_ADDR_TRIG	通道 4 的 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x130	CH4_AL3_CTRL	通道 4 的 CTRL 寄存器别名
0x134	CH4_AL3_WRITE_ADDR	通道 4 的 WRITE_ADDR 寄存器别名
0x138	CH4_AL3_TRANS_COUNT	通道 4 的 TRANS_COUNT 寄存器别名
0x13c	CH4_AL3_READ_ADDR_TRIG	通道 4 的 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x140	CH5_READ_ADDR	DMA 通道 5 读取地址指针
0x144	CH5_WRITE_ADDR	DMA 通道 5 写入地址指针
0x148	CH5_TRANS_COUNT	DMA 通道 5 传输计数
0x14c	CH5_CTRL_TRIG	DMA 通道 5 控制与状态
0x150	CH5_AL1_CTRL	通道 5 CTRL 寄存器别名
0x154	CH5_AL1_READ_ADDR	通道 5 READ_ADDR 寄存器别名
0x158	CH5_AL1_WRITE_ADDR	通道 5 WRITE_ADDR 寄存器别名
0x15c	CH5_AL1_TRANS_COUNT_TRIG	通道 5 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x160	CH5_AL2_CTRL	通道 5 CTRL 寄存器别名
0x164	CH5_AL2_TRANS_COUNT	通道 5 TRANS_COUNT 寄存器别名
0x168	CH5_AL2_READ_ADDR	通道 5 READ_ADDR 寄存器别名
0x16c	CH5_AL2_WRITE_ADDR_TRIG	通道 5 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x170	CH5_AL3_CTRL	通道 5 CTRL 寄存器别名
0x174	CH5_AL3_WRITE_ADDR	通道 5 WRITE_ADDR 寄存器别名
0x178	CH5_AL3_TRANS_COUNT	通道 5 TRANS_COUNT 寄存器别名

Offset	Name	Info
0x17c	CH5_AL3_READ_ADDR_TRIG	Alias for channel 5 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x180	CH6_READ_ADDR	DMA Channel 6 Read Address pointer
0x184	CH6_WRITE_ADDR	DMA Channel 6 Write Address pointer
0x188	CH6_TRANS_COUNT	DMA Channel 6 Transfer Count
0x18c	CH6_CTRL_TRIG	DMA Channel 6 Control and Status
0x190	CH6_AL1_CTRL	Alias for channel 6 CTRL register
0x194	CH6_AL1_READ_ADDR	Alias for channel 6 READ_ADDR register
0x198	CH6_AL1_WRITE_ADDR	Alias for channel 6 WRITE_ADDR register
0x19c	CH6_AL1_TRANS_COUNT_TRIG	Alias for channel 6 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x1a0	CH6_AL2_CTRL	Alias for channel 6 CTRL register
0x1a4	CH6_AL2_TRANS_COUNT	Alias for channel 6 TRANS_COUNT register
0x1a8	CH6_AL2_READ_ADDR	Alias for channel 6 READ_ADDR register
0x1ac	CH6_AL2_WRITE_ADDR_TRIG	Alias for channel 6 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x1b0	CH6_AL3_CTRL	Alias for channel 6 CTRL register
0x1b4	CH6_AL3_WRITE_ADDR	Alias for channel 6 WRITE_ADDR register
0x1b8	CH6_AL3_TRANS_COUNT	Alias for channel 6 TRANS_COUNT register
0x1bc	CH6_AL3_READ_ADDR_TRIG	Alias for channel 6 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x1c0	CH7_READ_ADDR	DMA Channel 7 Read Address pointer
0x1c4	CH7_WRITE_ADDR	DMA Channel 7 Write Address pointer
0x1c8	CH7_TRANS_COUNT	DMA Channel 7 Transfer Count
0x1cc	CH7_CTRL_TRIG	DMA Channel 7 Control and Status
0x1d0	CH7_AL1_CTRL	Alias for channel 7 CTRL register
0x1d4	CH7_AL1_READ_ADDR	Alias for channel 7 READ_ADDR register
0x1d8	CH7_AL1_WRITE_ADDR	Alias for channel 7 WRITE_ADDR register
0x1dc	CH7_AL1_TRANS_COUNT_TRIG	Alias for channel 7 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x1e0	CH7_AL2_CTRL	Alias for channel 7 CTRL register
0x1e4	CH7_AL2_TRANS_COUNT	Alias for channel 7 TRANS_COUNT register
0x1e8	CH7_AL2_READ_ADDR	Alias for channel 7 READ_ADDR register

偏移量	名称	说明
0x17c	CH5_AL3_READ_ADDR_TRIG	通道5 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x180	CH6_READ_ADDR	DMA通道6读取地址指针
0x184	CH6_WRITE_ADDR	DMA通道6写入地址指针
0x188	CH6_TRANS_COUNT	DMA通道6传输计数
0x18c	CH6_CTRL_TRIG	DMA通道6控制与状态
0x190	CH6_AL1_CTRL	通道6 CTRL寄存器别名
0x194	CH6_AL1_READ_ADDR	通道6 READ_ADDR寄存器别名
0x198	CH6_AL1_WRITE_ADDR	通道6 WRITE_ADDR寄存器别名
0x19c	CH6_AL1_TRANS_COUNT_TRIG	通道6 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1a0	CH6_AL2_CTRL	通道6 CTRL寄存器别名
0x1a4	CH6_AL2_TRANS_COUNT	通道6 TRANS_COUNT寄存器别名
0x1a8	CH6_AL2_READ_ADDR	通道6 READ_ADDR寄存器别名
0x1ac	CH6_AL2_WRITE_ADDR_TRIG	通道6 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1b0	CH6_AL3_CTRL	通道6 CTRL寄存器别名
0x1b4	CH6_AL3_WRITE_ADDR	通道6 WRITE_ADDR寄存器别名
0x1b8	CH6_AL3_TRANS_COUNT	通道6 TRANS_COUNT寄存器别名
0x1bc	CH6_AL3_READ_ADDR_TRIG	通道6 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1c0	CH7_READ_ADDR	DMA通道7读取地址指针
0x1c4	CH7_WRITE_ADDR	DMA通道7写入地址指针
0x1c8	CH7_TRANS_COUNT	DMA通道7传输计数
0x1cc	CH7_CTRL_TRIG	DMA通道7控制与状态
0x1d0	CH7_AL1_CTRL	通道7 CTRL寄存器别名
0x1d4	CH7_AL1_READ_ADDR	通道7 READ_ADDR寄存器别名
0x1d8	CH7_AL1_WRITE_ADDR	通道7 WRITE_ADDR寄存器别名
0x1dc	CH7_AL1_TRANS_COUNT_TRIG	通道7 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1e0	CH7_AL2_CTRL	通道7 CTRL寄存器别名
0x1e4	CH7_AL2_TRANS_COUNT	通道7 TRANS_COUNT寄存器别名
0x1e8	CH7_AL2_READ_ADDR	通道7 READ_ADDR寄存器别名

Offset	Name	Info
0x1ec	CH7_AL2_WRITE_ADDR_TRIG	Alias for channel 7 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x1f0	CH7_AL3_CTRL	Alias for channel 7 CTRL register
0x1f4	CH7_AL3_WRITE_ADDR	Alias for channel 7 WRITE_ADDR register
0x1f8	CH7_AL3_TRANS_COUNT	Alias for channel 7 TRANS_COUNT register
0x1fc	CH7_AL3_READ_ADDR_TRIG	Alias for channel 7 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x200	CH8_READ_ADDR	DMA Channel 8 Read Address pointer
0x204	CH8_WRITE_ADDR	DMA Channel 8 Write Address pointer
0x208	CH8_TRANS_COUNT	DMA Channel 8 Transfer Count
0x20c	CH8_CTRL_TRIG	DMA Channel 8 Control and Status
0x210	CH8_AL1_CTRL	Alias for channel 8 CTRL register
0x214	CH8_AL1_READ_ADDR	Alias for channel 8 READ_ADDR register
0x218	CH8_AL1_WRITE_ADDR	Alias for channel 8 WRITE_ADDR register
0x21c	CH8_AL1_TRANS_COUNT_TRIG	Alias for channel 8 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x220	CH8_AL2_CTRL	Alias for channel 8 CTRL register
0x224	CH8_AL2_TRANS_COUNT	Alias for channel 8 TRANS_COUNT register
0x228	CH8_AL2_READ_ADDR	Alias for channel 8 READ_ADDR register
0x22c	CH8_AL2_WRITE_ADDR_TRIG	Alias for channel 8 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x230	CH8_AL3_CTRL	Alias for channel 8 CTRL register
0x234	CH8_AL3_WRITE_ADDR	Alias for channel 8 WRITE_ADDR register
0x238	CH8_AL3_TRANS_COUNT	Alias for channel 8 TRANS_COUNT register
0x23c	CH8_AL3_READ_ADDR_TRIG	Alias for channel 8 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x240	CH9_READ_ADDR	DMA Channel 9 Read Address pointer
0x244	CH9_WRITE_ADDR	DMA Channel 9 Write Address pointer
0x248	CH9_TRANS_COUNT	DMA Channel 9 Transfer Count
0x24c	CH9_CTRL_TRIG	DMA Channel 9 Control and Status
0x250	CH9_AL1_CTRL	Alias for channel 9 CTRL register
0x254	CH9_AL1_READ_ADDR	Alias for channel 9 READ_ADDR register
0x258	CH9_AL1_WRITE_ADDR	Alias for channel 9 WRITE_ADDR register

偏移量	名称	说明
0x1ec	CH7_AL2_WRITE_ADDR_TRIG	通道7 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x1f0	CH7_AL3_CTRL	通道7 CTRL寄存器别名
0x1f4	CH7_AL3_WRITE_ADDR	通道7 WRITE_ADDR寄存器别名
0x1f8	CH7_AL3_TRANS_COUNT	通道7 TRANS_COUNT寄存器别名
0x1fc	CH7_AL3_READ_ADDR_TRIG	通道7 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x200	CH8_READ_ADDR	DMA 通道 8 读取地址指针
0x204	CH8_WRITE_ADDR	DMA 通道 8 写入地址指针
0x208	CH8_TRANS_COUNT	DMA 通道 8 传输计数
0x20c	CH8_CTRL_TRIG	DMA 通道 8 控制与状态寄存器
0x210	CH8_AL1_CTRL	通道 8 CTRL 寄存器别名
0x214	CH8_AL1_READ_ADDR	通道 8 READ_ADDR 寄存器别名
0x218	CH8_AL1_WRITE_ADDR	通道 8 WRITE_ADDR 寄存器别名
0x21c	CH8_AL1_TRANS_COUNT_TRIG	通道 8 TRANS_COUNT 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x220	CH8_AL2_CTRL	通道 8 CTRL 寄存器别名
0x224	CH8_AL2_TRANS_COUNT	通道 8 TRANS_COUNT 寄存器别名
0x228	CH8_AL2_READ_ADDR	通道 8 READ_ADDR 寄存器别名
0x22c	CH8_AL2_WRITE_ADDR_TRIG	通道 8 WRITE_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x230	CH8_AL3_CTRL	通道 8 CTRL 寄存器别名
0x234	CH8_AL3_WRITE_ADDR	通道 8 WRITE_ADDR 寄存器别名
0x238	CH8_AL3_TRANS_COUNT	通道 8 TRANS_COUNT 寄存器别名
0x23c	CH8_AL3_READ_ADDR_TRIG	通道 8 READ_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x240	CH9_READ_ADDR	DMA 通道9 读地址指针
0x244	CH9_WRITE_ADDR	DMA 通道9 写地址指针
0x248	CH9_TRANS_COUNT	DMA 通道9 传输计数
0x24c	CH9_CTRL_TRIG	DMA 通道9 控制与状态
0x250	CH9_AL1_CTRL	通道9 CTRL 寄存器的别名
0x254	CH9_AL1_READ_ADDR	通道9 READ_ADDR寄存器别名
0x258	CH9_AL1_WRITE_ADDR	通道9 WRITE_ADDR寄存器别名

Offset	Name	Info
0x25c	CH9_AL1_TRANS_COUNT_TRIG	Alias for channel 9 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x260	CH9_AL2_CTRL	Alias for channel 9 CTRL register
0x264	CH9_AL2_TRANS_COUNT	Alias for channel 9 TRANS_COUNT register
0x268	CH9_AL2_READ_ADDR	Alias for channel 9 READ_ADDR register
0x26c	CH9_AL2_WRITE_ADDR_TRIG	Alias for channel 9 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x270	CH9_AL3_CTRL	Alias for channel 9 CTRL register
0x274	CH9_AL3_WRITE_ADDR	Alias for channel 9 WRITE_ADDR register
0x278	CH9_AL3_TRANS_COUNT	Alias for channel 9 TRANS_COUNT register
0x27c	CH9_AL3_READ_ADDR_TRIG	Alias for channel 9 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x280	CH10_READ_ADDR	DMA Channel 10 Read Address pointer
0x284	CH10_WRITE_ADDR	DMA Channel 10 Write Address pointer
0x288	CH10_TRANS_COUNT	DMA Channel 10 Transfer Count
0x28c	CH10_CTRL_TRIG	DMA Channel 10 Control and Status
0x290	CH10_AL1_CTRL	Alias for channel 10 CTRL register
0x294	CH10_AL1_READ_ADDR	Alias for channel 10 READ_ADDR register
0x298	CH10_AL1_WRITE_ADDR	Alias for channel 10 WRITE_ADDR register
0x29c	CH10_AL1_TRANS_COUNT_TRIG	Alias for channel 10 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x2a0	CH10_AL2_CTRL	Alias for channel 10 CTRL register
0x2a4	CH10_AL2_TRANS_COUNT	Alias for channel 10 TRANS_COUNT register
0x2a8	CH10_AL2_READ_ADDR	Alias for channel 10 READ_ADDR register
0x2ac	CH10_AL2_WRITE_ADDR_TRIG	Alias for channel 10 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x2b0	CH10_AL3_CTRL	Alias for channel 10 CTRL register
0x2b4	CH10_AL3_WRITE_ADDR	Alias for channel 10 WRITE_ADDR register
0x2b8	CH10_AL3_TRANS_COUNT	Alias for channel 10 TRANS_COUNT register
0x2bc	CH10_AL3_READ_ADDR_TRIG	Alias for channel 10 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x2c0	CH11_READ_ADDR	DMA Channel 11 Read Address pointer
0x2c4	CH11_WRITE_ADDR	DMA Channel 11 Write Address pointer
0x2c8	CH11_TRANS_COUNT	DMA Channel 11 Transfer Count

偏移量	名称	说明
0x25c	CH9_AL1_TRANS_COUNT_TRIG	通道9 TRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x260	CH9_AL2_CTRL	通道9 CTRL 寄存器的别名
0x264	CH9_AL2_TRANS_COUNT	通道9 TRANS_COUNT寄存器别名
0x268	CH9_AL2_READ_ADDR	通道9 READ_ADDR寄存器别名
0x26c	CH9_AL2_WRITE_ADDR_TRIG	通道9 WRITE_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x270	CH9_AL3_CTRL	通道9 CTRL 寄存器的别名
0x274	CH9_AL3_WRITE_ADDR	通道9 WRITE_ADDR寄存器别名
0x278	CH9_AL3_TRANS_COUNT	通道9 TRANS_COUNT寄存器别名
0x27c	CH9_AL3_READ_ADDR_TRIG	通道9 READ_ADDR寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x280	CH10_READ_ADDR	DMA通道10读地址指针
0x284	CH10_WRITE_ADDR	DMA通道10写地址指针
0x288	CH10_TRANS_COUNT	DMA 通道 10 传输计数
0x28c	CH10_CTRL_TRIG	DMA 通道 10 控制与状态
0x290	CH10_AL1_CTRL	通道 10 CTRL 寄存器别名
0x294	CH10_AL1_READ_ADDR	通道 10 READ_ADDR 寄存器别名
0x298	CH10_AL1_WRITE_ADDR	通道 10 WRITE_ADDR 寄存器别名
0x29c	CH10_AL1_TRANS_COUNT_TRIG	通道 10 TRANS_COUNT 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2a0	CH10_AL2_CTRL	通道 10 CTRL 寄存器别名
0x2a4	CH10_AL2_TRANS_COUNT	通道 10 TRANS_COUNT 寄存器别名
0x2a8	CH10_AL2_READ_ADDR	通道 10 READ_ADDR 寄存器别名
0x2ac	CH10_AL2_WRITE_ADDR_TRIG	通道 10 WRITE_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2b0	CH10_AL3_CTRL	通道 10 CTRL 寄存器别名
0x2b4	CH10_AL3_WRITE_ADDR	通道 10 WRITE_ADDR 寄存器别名
0x2b8	CH10_AL3_TRANS_COUNT	通道 10 TRANS_COUNT 寄存器别名
0x2bc	CH10_AL3_READ_ADDR_TRIG	通道 10 READ_ADDR 寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。
0x2c0	CH11_READ_ADDR	DMA通道11读取地址指针
0x2c4	CH11_WRITE_ADDR	DMA通道11写入地址指针
0x2c8	CH11_TRANS_COUNT	DMA通道11传输计数

Offset	Name	Info
0x2cc	CH11_CTRL_TRIG	DMA Channel 11 Control and Status
0x2d0	CH11_AL1_CTRL	Alias for channel 11 CTRL register
0x2d4	CH11_AL1_READ_ADDR	Alias for channel 11 READ_ADDR register
0x2d8	CH11_AL1_WRITE_ADDR	Alias for channel 11 WRITE_ADDR register
0x2dc	CH11_AL1_TRANS_COUNT_TRIG	Alias for channel 11 TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x2e0	CH11_AL2_CTRL	Alias for channel 11 CTRL register
0x2e4	CH11_AL2_TRANS_COUNT	Alias for channel 11 TRANS_COUNT register
0x2e8	CH11_AL2_READ_ADDR	Alias for channel 11 READ_ADDR register
0x2ec	CH11_AL2_WRITE_ADDR_TRIG	Alias for channel 11 WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x2f0	CH11_AL3_CTRL	Alias for channel 11 CTRL register
0x2f4	CH11_AL3_WRITE_ADDR	Alias for channel 11 WRITE_ADDR register
0x2f8	CH11_AL3_TRANS_COUNT	Alias for channel 11 TRANS_COUNT register
0xfc	CH11_AL3_READ_ADDR_TRIG	Alias for channel 11 READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.
0x400	INTR	Interrupt Status (raw)
0x404	INTE0	Interrupt Enables for IRQ 0
0x408	INTF0	Force Interrupts
0x40c	INTS0	Interrupt Status for IRQ 0
0x414	INTE1	Interrupt Enables for IRQ 1
0x418	INTF1	Force Interrupts for IRQ 1
0x41c	INTS1	Interrupt Status (masked) for IRQ 1
0x420	TIMER0	Pacing (X/Y) Fractional Timer The pacing timer produces TREQ assertions at a rate set by $((X/Y) * sys_clk)$. This equation is evaluated every sys_clk cycles and therefore can only generate TREQs at a rate of 1 per sys_clk (i.e. permanent TREQ) or less.
0x424	TIMER1	Pacing (X/Y) Fractional Timer The pacing timer produces TREQ assertions at a rate set by $((X/Y) * sys_clk)$. This equation is evaluated every sys_clk cycles and therefore can only generate TREQs at a rate of 1 per sys_clk (i.e. permanent TREQ) or less.
0x428	TIMER2	Pacing (X/Y) Fractional Timer The pacing timer produces TREQ assertions at a rate set by $((X/Y) * sys_clk)$. This equation is evaluated every sys_clk cycles and therefore can only generate TREQs at a rate of 1 per sys_clk (i.e. permanent TREQ) or less.

偏移量	名称	说明
0x2cc	CH11_CTRL_TRIG	DMA通道11控制与状态
0x2d0	CH11_AL1_CTRL	通道11 CTRL 寄存器别名
0x2d4	CH11_AL1_READ_ADDR	通道11 READ_ADDR 寄存器别名
0x2d8	CH11_AL1_WRITE_ADDR	通道11 WRITE_ADDR 寄存器别名
0x2dc	CH11_AL1_TRANS_COUNT_TRIG	通道11 TRANS_COUNT 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x2e0	CH11_AL2_CTRL	通道11 CTRL 寄存器别名
0x2e4	CH11_AL2_TRANS_COUNT	通道11 TRANS_COUNT 寄存器别名
0x2e8	CH11_AL2_READ_ADDR	通道11 READ_ADDR 寄存器别名
0x2ec	CH11_AL2_WRITE_ADDR_TRIG	通道11 WRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x2f0	CH11_AL3_CTRL	通道11 CTRL 寄存器别名
0x2f4	CH11_AL3_WRITE_ADDR	通道11 WRITE_ADDR 寄存器别名
0x2f8	CH11_AL3_TRANS_COUNT	通道11 TRANS_COUNT 寄存器别名
0x2fc	CH11_AL3_READ_ADDR_TRIG	通道11 READ_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。
0x400	INTR	中断状态 (原始)
0x404	INTE0	IRQ 0中断使能
0x408	INTF0	强制中断
0x40c	INTS0	IRQ 0 的中断状态
0x414	INTE1	IRQ 1 的中断使能
0x418	INTF1	IRQ 1 的强制中断
0x41c	INTS1	IRQ 1 的中断状态 (屏蔽)
0x420	TIMER0	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。
0x424	TIMER1	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。
0x428	TIMER2	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次 (即持续 TREQ) 或更低速率产生 TREQ。

Offset	Name	Info
0x42c	TIMER3	Pacing (X/Y) Fractional Timer The pacing timer produces TREQ assertions at a rate set by $((X/Y) * \text{sys_clk})$. This equation is evaluated every sys_clk cycles and therefore can only generate TREQs at a rate of 1 per sys_clk (i.e. permanent TREQ) or less.
0x430	MULTI_CHAN_TRIGGER	Trigger one or more channels simultaneously
0x434	SNIFF_CTRL	Sniffer Control
0x438	SNIFF_DATA	Data accumulator for sniff hardware
0x440	FIFO_LEVELS	Debug RAF, WAF, TDF levels
0x444	CHAN_ABORT	Abort an in-progress transfer sequence on one or more channels
0x448	N_CHANNELS	The number of channels this DMA instance is equipped with. This DMA supports up to 16 hardware channels, but can be configured with as few as one, to minimise silicon area.
0x800	CH0_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x804	CH0_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x840	CH1_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x844	CH1_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x880	CH2_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x884	CH2_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x8c0	CH3_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x8c4	CH3_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x900	CH4_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x904	CH4_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer

偏移量	名称	说明
0x42c	TIMER3	节拍 (X/Y) 分数定时器 该节拍定时器以 $((X/Y) * sys_clk)$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次（即持续 TREQ）或更低速率产生 TREQ。
0x430	MULTI_CHAN_TRIGGER	同时触发一个或多个通道
0x434	SNIFF_CTRL	嗅探器控制
0x438	SNIFF_DATA	嗅探硬件数据累加器
0x440	FIFO_LEVELS	调试 RAF、WAF、TDF 级别
0x444	CHAN_ABORT	中止一个或多个通道上的正在进行传输序列
0x448	N_CHANNELS	该 DMA 实例配置的通道数量。 该 DMA 最多支持 16 个硬件通道，但可配置为最少 1 个，以减少芯片面积。
0x800	CH0_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x804	CH0_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x840	CH1_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x844	CH1_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x880	CH2_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x884	CH2_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x8c0	CH3_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x8c4	CH3_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x900	CH4_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x904	CH4_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度

Offset	Name	Info
0x940	CH5_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x944	CH5_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x980	CH6_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x984	CH6_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0x9c0	CH7_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0x9c4	CH7_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0xa00	CH8_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0xa04	CH8_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0xa40	CH9_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0xa44	CH9_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0xa80	CH10_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0xa84	CH10_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer
0xac0	CH11_DBG_CTDREQ	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.
0xac4	CH11_DBG_TCR	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer

DMA: CH0_READ_ADDR, CH1_READ_ADDR, ..., CH10_READ_ADDR, CH11_READ_ADDR Registers

Offsets: 0x000, 0x040, ..., 0x280, 0x2c0

偏移量	名称	说明
0x940	CH5_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x944	CH5_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x980	CH6_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x984	CH6_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0x9c0	CH7_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0x9c4	CH7_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa00	CH8_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa04	CH8_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa40	CH9_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa44	CH9_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xa80	CH10_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xa84	CH10_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度
0xac0	CH11_DBG_CTDREQ	读取：获取通道 DREQ 计数器（即 DMA 预期可对外设进行的访问次数，确保无溢出/下溢）。写入任意值：清除计数器，并使通道重新发起 DREQ 握手。
0xac4	CH11_DBG_TCR	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度

DMA: CH0_READ_ADDR, CH1_READ_ADDR, ..., CH10_READ_ADDR, CH11_READ_ADDR 寄存器

偏移量: 0x000, 0x040, ..., 0x280, 0x2c0

Description

DMA Channel N Read Address pointer

Table 121.
CH0_READ_ADDR,
CH1_READ_ADDR, ...
CH10_READ_ADDR,
CH11_READ_ADDR
Registers

Bits	Description	Type	Reset
31:0	This register updates automatically each time a read completes. The current value is the next address to be read by this channel.	RW	0x00000000

DMA: CH0_WRITE_ADDR, CH1_WRITE_ADDR, ..., CH10_WRITE_ADDR, CH11_WRITE_ADDR Registers**Offsets:** 0x004, 0x044, ..., 0x284, 0x2c4**Description**

DMA Channel N Write Address pointer

Table 122.
CH0_WRITE_ADDR,
CH1_WRITE_ADDR, ...
CH10_WRITE_ADDR,
CH11_WRITE_ADDR
Registers

Bits	Description	Type	Reset
31:0	This register updates automatically each time a write completes. The current value is the next address to be written by this channel.	RW	0x00000000

DMA: CH0_TRANS_COUNT, CH1_TRANS_COUNT, ..., CH10_TRANS_COUNT, CH11_TRANS_COUNT Registers**Offsets:** 0x008, 0x048, ..., 0x288, 0x2c8**Description**

DMA Channel N Transfer Count

Table 123.
CH0_TRANS_COUNT,
CH1_TRANS_COUNT,
...
CH10_TRANS_COUNT,
CH11_TRANS_COUNT
Registers

Bits	Description	Type	Reset
31:0	<p>Program the number of bus transfers a channel will perform before halting. Note that, if transfers are larger than one byte in size, this is not equal to the number of bytes transferred (see CTRL_DATA_SIZE).</p> <p>When the channel is active, reading this register shows the number of transfers remaining, updating automatically each time a write transfer completes.</p> <p>Writing this register sets the RELOAD value for the transfer counter. Each time this channel is triggered, the RELOAD value is copied into the live transfer counter. The channel can be started multiple times, and will perform the same number of transfers each time, as programmed by most recent write.</p> <p>The RELOAD value can be observed at CHx_DBG_TCR. If TRANS_COUNT is used as a trigger, the written value is used immediately as the length of the new transfer sequence, as well as being written to RELOAD.</p>	RW	0x00000000

DMA: CH0_CTRL_TRIG, CH1_CTRL_TRIG, ..., CH10_CTRL_TRIG, CH11_CTRL_TRIG Registers**Offsets:** 0x00c, 0x04c, ..., 0x28c, 0x2cc**Description**

DMA Channel N Control and Status

描述

DMA 通道 N 读地址指针

表 121
 CH0_READ_ADDR,
 CH1_READ_ADDR, ...
 CH10_READ_ADDR,
 CH11_READ_ADDR
 寄存器

位	描述	类型	复位值
31:0	该寄存器在每次读完成时自动更新。当前值为该通道下次待读的地址。	读写	0x00000000

DMA: CH0_WRITE_ADDR, CH1_WRITE_ADDR, ..., CH10_WRITE_ADDR, CH11_WRITE_ADDR 寄存器

偏移量: 0x004, 0x044, ..., 0x284, 0x2c4

描述

DMA 通道 N 写地址指针

表 122。
 CH0_WRITE_ADDR,
 CH1_WRITE_ADDR, ...
 CH10_WRITE_ADDR,
 CH11_WRITE_ADDR
 寄存器

位	描述	类型	复位值
31:0	该寄存器在每次写入完成后自动更新。当前值为该通道将要写入的下一个地址。	读写	0x00000000

DMA: CH0_TRANS_COUNT, CH1_TRANS_COUNT, ..., CH10_TRANS_COUNT, CH11_TRANS_COUNT 寄存器

偏移量: 0x008, 0x048, ..., 0x288, 0x2c8

描述

DMA 通道 N 传输计数

表 123。
 CH0_TRANS_COUNT,
 CH1_TRANS_COUNT,
 ...
 CH10_TRANS_COUNT,
 CH11_TRANS_COUNT
 寄存器

位	描述	类型	复位值
31:0	<p>设置通道在停止前将执行的总线传输次数。</p> <p>注意, 若传输大小超过一个字节, 该数值不等于传输的字节数 (参见CTRL_DATA_SIZE)。</p> <p>当通道处于活动状态时, 读取该寄存器显示剩余传输次数, 且在每次写传输完成后自动更新。</p> <p>写入此寄存器将设置传输计数器的 RELOAD 值。每次触发该通道时, RELOAD 值将被复制到实时传输计数器。该通道可多次启动, 每次启动将执行相同次数的传输, 次数由最近一次写入的值确定。</p> <p>RELOAD 值可通过 CHx_DBG_TCR 进行观察。若以 TRANS_COUNT 作为触发条件, 写入值将立即用作新传输序列的长度, 并写入 RELOAD。</p>	读写	0x00000000

DMA: CH0_CTRL_TRIG, CH1_CTRL_TRIG, ..., CH10_CTRL_TRIG, CH11_CTRL_TRIG 寄存器

偏移量: 0x00c, 0x04c, ..., 0x28c, 0x2cc

描述

DMA 通道 N 控制与状态寄存器

Table 124.
CH0_CTRL_TRIG,
CH1_CTRL_TRIG, ...
CH10_CTRL_TRIG,
CH11_CTRL_TRIG
Registers

Bits	Description	Type	Reset
31	AHB_ERROR: Logical OR of the READ_ERROR and WRITE_ERROR flags. The channel halts when it encounters any bus error, and always raises its channel IRQ flag.	RO	0x0
30	READ_ERROR: If 1, the channel received a read bus error. Write one to clear. READ_ADDR shows the approximate address where the bus error was encountered (will not be earlier, or more than 3 transfers later)	WC	0x0
29	WRITE_ERROR: If 1, the channel received a write bus error. Write one to clear. WRITE_ADDR shows the approximate address where the bus error was encountered (will not be earlier, or more than 5 transfers later)	WC	0x0
28:25	Reserved.	-	-
24	BUSY: This flag goes high when the channel starts a new transfer sequence, and low when the last transfer of that sequence completes. Clearing EN while BUSY is high pauses the channel, and BUSY will stay high while paused. To terminate a sequence early (and clear the BUSY flag), see CHAN_ABORT.	RO	0x0
23	SNIFF_EN: If 1, this channel's data transfers are visible to the sniff hardware, and each transfer will advance the state of the checksum. This only applies if the sniff hardware is enabled, and has this channel selected. This allows checksum to be enabled or disabled on a per-control-block basis.	RW	0x0
22	BSWAP: Apply byte-swap transformation to DMA data. For byte data, this has no effect. For halfword data, the two bytes of each halfword are swapped. For word data, the four bytes of each word are swapped to reverse order.	RW	0x0
21	IRQ QUIET: In QUIET mode, the channel does not generate IRQs at the end of every transfer block. Instead, an IRQ is raised when NULL is written to a trigger register, indicating the end of a control block chain. This reduces the number of interrupts to be serviced by the CPU when transferring a DMA chain of many small control blocks.	RW	0x0
20:15	TREQ_SEL: Select a Transfer Request signal. The channel uses the transfer request signal to pace its data transfer rate. Sources for TREQ signals are internal (TIMERS) or external (DREQ, a Data Request from the system). 0x0 to 0x3a → select DREQ n as TREQ	RW	0x00
	Enumerated values:		
	0x3b → TIMER0: Select Timer 0 as TREQ		
	0x3c → TIMER1: Select Timer 1 as TREQ		
	0x3d → TIMER2: Select Timer 2 as TREQ (Optional)		
	0x3e → TIMER3: Select Timer 3 as TREQ (Optional)		
	0x3f → PERMANENT: Permanent request, for unpaced transfers.		
14:11	CHAIN_TO: When this channel completes, it will trigger the channel indicated by CHAIN_TO. Disable by setting CHAIN_TO = (this channel).	RW	0x0
10	RING_SEL: Select whether RING_SIZE applies to read or write addresses. If 0, read addresses are wrapped on a (1 << RING_SIZE) boundary. If 1, write addresses are wrapped.	RW	0x0

表 124。
 $CH0_CTRL_TRIG$
 $\backslash CH1_CTRL_TRIG$
 $\backslash \dots CH10_CTR$
 $L_TRIG, CH11_C$
 TRL_TRIG 寄存器

位	描述	类型	复位值
31	AHB_ERROR: READ_ERROR 与 WRITE_ERROR 标志的逻辑或。当遇到任意总线错误时，通道将停止，并始终触发其通道 IRQ 标志。	只读	0x0
30	READ_ERROR: 若为1，表示通道接收到读总线错误。写入1可清除该标志。READ_ADDR 显示遇到总线错误的大致地址（不会早于实际发生地址，且不会晚于3次传输后）。	WC	0x0
29	WRITE_ERROR: 若为1，表示通道接收到写总线错误。写入1可清除该标志。WRITE_ADDR 显示遇到总线错误的大致地址（不会早于实际发生地址，且不会晚于5次传输后）。	WC	0x0
28:25	保留。	-	-
24	BUSY: 当通道开始新的传输序列时，该标志置高；当该序列的最后一次传输完成时，标志置低。在 BUSY 标志为高时清除 EN 会暂停通道，且暂停期间 BUSY 标志保持为高。 如需提前终止序列（并清除 BUSY 标志），请参见 CHAN_ABORT。	只读	0x0
23	SNIFF_EN: 若为1，本通道的数据传输对嗅探硬件可见，且每次传输将使校验和状态前进。此项仅在嗅探硬件启用且选择了该通道时适用。 此设置允许基于每个控制块启用或禁用校验和功能。	读写	0x0
22	BSWAP: 对DMA数据应用字节交换变换。 对于字节数据，此设置无效。对于半字数据，每个半字的两个字节将被交换。对于字数据，每个字的四个字节将被交换以实现逆序。	读写	0x0
21	IRQ QUIET: 在静默（QUIET）模式下，该通道在每个传输块结束时不会产生中断请求（IRQ）。相反，当触发寄存器写入NULL时，会产生中断请求，表明控制块链已结束。 在传输包含多个小控制块的DMA链时，该机制减少了CPU需处理的中断次数。	读写	0x0
20:15	TREQ_SEL: 选择传输请求信号。 该通道使用传输请求信号以调整数据传输速率。 传输请求信号的来源包括内部（定时器）或外部（系统发出的数据请求DR EQ）。 0x0 到 0x3a → 选择 DREQ n 作为 TREQ	读写	0x00
	枚举值：		
	0x3b → TIMER0：选择定时器 0 作为 TREQ		
	0x3c → TIMER1：选择定时器 1 作为 TREQ		
	0x3d → TIMER2：选择定时器 2 作为 TREQ（可选）		
	0x3e → TIMER3：选择定时器 3 作为 TREQ（可选）		
	0x3f → PERMANENT：永久请求，适用于无节奏传输。		
14:11	CHAIN_TO: 当此通道完成时，将触发由 CHAIN_TO 指示的通道。通过设置 C HAIN_TO = (本通道) 来禁用。	读写	0x0
10	RING_SEL: 选择 RING_SIZE 应用于读地址还是写地址。 若为 0，则读地址围绕 ($1 \ll RING_SIZE$) 边界回绕。若为 1，则写地址回绕。	读写	0x0

Bits	Description	Type	Reset
9:6	RING_SIZE: Size of address wrap region. If 0, don't wrap. For values n > 0, only the lower n bits of the address will change. This wraps the address on a $(1 \ll n)$ byte boundary, facilitating access to naturally-aligned ring buffers. Ring sizes between 2 and 32768 bytes are possible. This can apply to either read or write addresses, based on value of RING_SEL.	RW	0x0
	Enumerated values:		
	0x0 → RING_NONE		
5	INCR_WRITE: If 1, the write address increments with each transfer. If 0, each write is directed to the same, initial address. Generally this should be disabled for memory-to-peripheral transfers.	RW	0x0
4	INCR_READ: If 1, the read address increments with each transfer. If 0, each read is directed to the same, initial address. Generally this should be disabled for peripheral-to-memory transfers.	RW	0x0
3:2	DATA_SIZE: Set the size of each bus transfer (byte/halfword/word). READ_ADDR and WRITE_ADDR advance by this amount (1/2/4 bytes) with each transfer.	RW	0x0
	Enumerated values:		
	0x0 → SIZE_BYTE		
	0x1 → SIZE_HALFWORD		
	0x2 → SIZE_WORD		
1	HIGH_PRIORITY: HIGH_PRIORITY gives a channel preferential treatment in issue scheduling: in each scheduling round, all high priority channels are considered first, and then only a single low priority channel, before returning to the high priority channels. This only affects the order in which the DMA schedules channels. The DMA's bus priority is not changed. If the DMA is not saturated then a low priority channel will see no loss of throughput.	RW	0x0
0	EN: DMA Channel Enable. When 1, the channel will respond to triggering events, which will cause it to become BUSY and start transferring data. When 0, the channel will ignore triggers, stop issuing transfers, and pause the current transfer sequence (i.e. BUSY will remain high if already high)	RW	0x0

DMA: CH0_AL1_CTRL, CH1_AL1_CTRL, ..., CH10_AL1_CTRL, CH11_AL1_CTRL Registers

Offsets: 0x010, 0x050, ..., 0x290, 0x2d0

位	描述	类型	复位值
9:6	<p>RING_SIZE: 地址回绕区域的大小。若为 0，则不回绕。对于 $n > 0$ 的值，仅地址的低 n 位会发生变化。该操作将地址限制在 $(1 \ll n)$ 字节的边界内，便于访问自然对齐的环形缓冲区。</p> <p>环形缓冲区的大小范围为 2 至 32768 字节。此功能可根据 RING_SEL 的取值，应用于读地址或写地址。</p>	读写	0x0
	枚举值：		
	0x0 → RING_NONE		
5	<p>INCR_WRITE: 若值为 1，写地址会在每次传输后递增。若值为 0，每次写入均指向相同的初始地址。</p> <p>通常对此功能应禁止，以适用于内存到外设的传输。</p>	读写	0x0
4	<p>INCR_READ: 若值为 1，读地址会在每次传输后递增。若值为 0，每次读取均指向相同的初始地址。</p> <p>通常对此功能应禁止，以适用于外设到内存的传输。</p>	读写	0x0
3:2	<p>DATA_SIZE: 设置每次总线传输的数据大小（字节／半字／字）。READ_ADDR 和 WRITE_ADDR 会在每次传输后，按此大小（1／2／4 字节）递增。</p>	读写	0x0
	枚举值：		
	0x0 → SIZE_BYTE		
	0x1 → SIZE_HALFWORD		
	0x2 → SIZE_WORD		
1	<p>HIGH_PRIORITY: HIGH_PRIORITY 使通道在请求调度中享有优先权：每个调度周期内，所有高优先级通道首先被考虑，然后仅调度一个低优先级通道，之后再返回高优先级通道。</p> <p>此设置仅影响 DMA 调度通道的顺序。DMA 的总线优先级保持不变。若 DMA 未达到饱和状态，低优先级通道的吞吐量不会受到影响。</p>	读写	0x0
0	<p>EN: DMA 通道使能。</p> <p>当该位为 1 时，通道响应触发事件，导致其进入 BUSY 状态并开始传输数据。当该位为 0 时，通道忽略触发，停止发出传输请求，并暂停当前传输序列（即若 BUSY 已处于高电平则保持该状态）。</p>	读写	0x0

DMA: CH0_AL1_CTRL, CH1_AL1_CTRL, ..., CH10_AL1_CTRL, CH11_AL1_CTRL 寄存器

偏移量: 0x010, 0x050, ..., 0x290, 0x2d0

Table 125.
`CH0_AL1_CTRL,`
`CH1_AL1_CTRL, ...`
`CH10_AL1_CTRL,`
`CH11_AL1_CTRL`
Registers

Bits	Description	Type	Reset
31:0	Alias for channel N CTRL register	RW	-

DMA: `CH0_AL1_READ_ADDR, CH1_AL1_READ_ADDR, ...`
`CH10_AL1_READ_ADDR, CH11_AL1_READ_ADDR Registers`

Offsets: 0x014, 0x054, ..., 0x294, 0x2d4

Table 126.
`CH0_AL1_READ_ADDR`
`, CH1_AL1_READ_ADDR`
`, ...`
`CH10_AL1_READ_ADDR`
`R, CH11_AL1_READ_ADDR`
Registers

Bits	Description	Type	Reset
31:0	Alias for channel N READ_ADDR register	RW	-

DMA: `CH0_AL1_WRITE_ADDR, CH1_AL1_WRITE_ADDR, ...`
`CH10_AL1_WRITE_ADDR, CH11_AL1_WRITE_ADDR Registers`

Offsets: 0x018, 0x058, ..., 0x298, 0x2d8

Table 127.
`CH0_AL1_WRITE_ADDR`
`R, CH1_AL1_WRITE_ADDR`
`R, ...`
`CH10_AL1_WRITE_ADDR`
`DR, CH11_AL1_WRITE_ADDR`
Registers

Bits	Description	Type	Reset
31:0	Alias for channel N WRITE_ADDR register	RW	-

DMA: `CH0_AL1_TRANS_COUNT_TRIG, CH1_AL1_TRANS_COUNT_TRIG, ...`
`CH10_AL1_TRANS_COUNT_TRIG, CH11_AL1_TRANS_COUNT_TRIG Registers`

Offsets: 0x01c, 0x05c, ..., 0x29c, 0x2dc

Table 128.
`CH0_AL1_TRANS_CO`
`NT_TRIG, CH1_AL1_TRANS_CO`
`NT_TRIG, ...`
`CH10_AL1_TRANS_CO`
`UNT_TRIG, CH11_AL1_TRANS_CO`
`UNT_TRIG Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N TRANS_COUNT register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.	RW	-

DMA: `CH0_AL2_CTRL, CH1_AL2_CTRL, ...`
`CH10_AL2_CTRL, CH11_AL2_CTRL Registers`

Offsets: 0x020, 0x060, ..., 0x2a0, 0x2e0

Table 129.
`CH0_AL2_CTRL,`
`CH1_AL2_CTRL, ...`
`CH10_AL2_CTRL,`
`CH11_AL2_CTRL`
Registers

Bits	Description	Type	Reset
31:0	Alias for channel N CTRL register	RW	-

DMA: `CH0_AL2_TRANS_COUNT, CH1_AL2_TRANS_COUNT, ...`
`CH10_AL2_TRANS_COUNT, CH11_AL2_TRANS_COUNT Registers`

Offsets: 0x024, 0x064, ..., 0x2a4, 0x2e4

Table 130.
`CH0_AL2_TRANS_CO`
`NT, CH1_AL2_TRANS_CO`
`NT, ...`
`CH10_AL2_TRANS_CO`
`UNT, CH11_AL2_TRANS_CO`
`UNT Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N TRANS_COUNT register	RW	-

DMA: `CH0_AL2_READ_ADDR, CH1_AL2_READ_ADDR, ...`
`CH10_AL2_READ_ADDR, CH11_AL2_READ_ADDR Registers`

Offsets: 0x028, 0x068, ..., 0x2a8, 0x2e8

表125。
CH0_AL1_CTRL
、 CH1_AL1_CTRL
、 ...、 CH10_AL1
_CTRL、 CH11_A
L1_CTRL寄存器

位	描述	类型	复位值
31:0	通道 NCTRL寄存器别名	读写	-

DMA: CH0_AL1_READ_ADDR, CH1_AL1_READ_ADDR, ..., CH10_AL1_READ_ADDR、 CH11_AL1_READ_ADDR寄存器

偏移量: 0x014, 0x054, ..., 0x294, 0x2d4

表126。
CH0_AL1_READ_ADDR
, CH1_AL1_READ_ADDR
,...
CH10_AL1_READ_ADDR
R,
CH11_AL1_READ_ADDR
R寄存器

位	描述	类型	复位值
31:0	通道 NREAD_ADDR寄存器别名	读写	-

DMA: CH0_AL1_WRITE_ADDR, CH1_AL1_WRITE_ADDR, ..., CH10_AL1_WRITE_ADDR、 CH11_AL1_WRITE_ADDR寄存器

偏移量: 0x018, 0x058, ..., 0x298, 0x2d8

表127。
CH0_AL1_WRITE_ADD
R,
CH1_AL1_WRITE_ADD
R,...
CH10_AL1_WRITE_ADD
DR,
CH11_AL1_WRITE_ADD
DR寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR寄存器别名	读写	-

DMA: CH0_AL1_TRANS_COUNT_TRIG, CH1_AL1_TRANS_COUNT_TRIG, ..., CH10_AL1_TRANS_COUNT_TRIG, CH11_AL1_TRANS_COUNT_TRIG寄存器

偏移量: 0x01c, 0x05c, ..., 0x29c, 0x2dc

表128。
CH0_AL1_TRANS_CO
NT_TRIG,
CH1_AL1_TRANS_CO
NT_TRIG, ...
CH10_AL1_TRANS_CO
UNT_TRIG,
CH11_AL1_TRANS_CO
UNT_TRIG寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT寄存器别名 这是一个触发寄存器（0xc）。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: CH0_AL2_CTRL, CH1_AL2_CTRL, ..., CH10_AL2_CTRL, CH11_AL2_CTRL寄存器

偏移: 0x020, 0x060, ..., 0x2a0, 0x2e0

表129。
CH0_AL2_CTRL
、 CH1_AL2_CTRL
、 ...、 CH10_AL2_CT
RL、 CH11_AL2_CT
寄存器

位	描述	类型	复位值
31:0	通道 NCTRL寄存器别名	读写	-

DMA: CH0_AL2_TRANS_COUNT、 CH1_AL2_TRANS_COUNT、 ...、 CH10_AL2_TRANS_COUNT、 CH11_AL2_TRANS_COUNT 寄存器

偏移: 0x024, 0x064, ..., 0x2a4, 0x2e4

表130。
CH0_AL2_TRANS_CO
UNT、 CH1_AL2
_TRANS_COUNT、 ...
、 CH10_AL2_T
RANS_COUNT、 CH11
_AL2_TRANS_C
OUNT寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT寄存器别名	读写	-

DMA: CH0_AL2_READ_ADDR, CH1_AL2_READ_ADDR, ..., CH10_AL2_READ_ADDR、 CH11_AL2_READ_ADDR 寄存器

偏移: 0x028, 0x068, ..., 0x2a8, 0x2e8

Table 131.
`CH0_AL2_READ_ADDR`,
`CH1_AL2_READ_ADDR`,
`...`
`CH10_AL2_READ_ADDR`,
`CH11_AL2_READ_ADDR Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N READ_ADDR register	RW	-

DMA: CH0_AL2_WRITE_ADDR_TRIG, CH1_AL2_WRITE_ADDR_TRIG, ..., CH10_AL2_WRITE_ADDR_TRIG, CH11_AL2_WRITE_ADDR_TRIG Registers

Offsets: 0x02c, 0x06c, ..., 0x2ac, 0x2ec

Table 132.
`CH0_AL2_WRITE_ADDR_TRIG`,
`CH1_AL2_WRITE_ADDR_TRIG`,
`...`
`CH10_AL2_WRITE_ADDR_TRIG`,
`CH11_AL2_WRITE_ADDR_TRIG Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N WRITE_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.	RW	-

DMA: CH0_AL3_CTRL, CH1_AL3_CTRL, ..., CH10_AL3_CTRL, CH11_AL3_CTRL Registers

Offsets: 0x030, 0x070, ..., 0x2b0, 0x2f0

Table 133.
`CH0_AL3_CTRL`,
`CH1_AL3_CTRL`,
`...`
`CH10_AL3_CTRL`,
`CH11_AL3_CTRL Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N CTRL register	RW	-

DMA: CH0_AL3_WRITE_ADDR, CH1_AL3_WRITE_ADDR, ..., CH10_AL3_WRITE_ADDR, CH11_AL3_WRITE_ADDR Registers

Offsets: 0x034, 0x074, ..., 0x2b4, 0x2f4

Table 134.
`CH0_AL3_WRITE_ADDR_R`,
`CH1_AL3_WRITE_ADDR_R`,
`...`
`CH10_AL3_WRITE_ADDR_DR`,
`CH11_AL3_WRITE_ADDR_DR Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N WRITE_ADDR register	RW	-

DMA: CH0_AL3_TRANS_COUNT, CH1_AL3_TRANS_COUNT, ..., CH10_AL3_TRANS_COUNT, CH11_AL3_TRANS_COUNT Registers

Offsets: 0x038, 0x078, ..., 0x2b8, 0x2f8

Table 135.
`CH0_AL3_TRANS_CNT_NT`,
`CH1_AL3_TRANS_CNT_NT`,
`...`
`CH10_AL3_TRANS_CNT_UNT`,
`CH11_AL3_TRANS_CNT_UNT Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N TRANS_COUNT register	RW	-

DMA: CH0_AL3_READ_ADDR_TRIG, CH1_AL3_READ_ADDR_TRIG, ..., CH10_AL3_READ_ADDR_TRIG, CH11_AL3_READ_ADDR_TRIG Registers

Offsets: 0x03c, 0x07c, ..., 0x2bc, 0x2fc

Table 136.
`CH0_AL3_READ_ADDR_TRIG`,
`CH1_AL3_READ_ADDR_TRIG`,
`...`
`CH10_AL3_READ_ADDR_RTRIG`,
`CH11_AL3_READ_ADDR_RTRIG Registers`

Bits	Description	Type	Reset
31:0	Alias for channel N READ_ADDR register This is a trigger register (0xc). Writing a nonzero value will reload the channel counter and start the channel.	RW	-

DMA: INTR Register

Offset: 0x400

Description

Interrupt Status (raw)

表131。
CH0_AL2_READ_ADDR
,
CH1_AL2_READ_ADDR
,

位	描述	类型	复位值
31:0	通道 NREAD_ADDR 寄存器别名	读写	-

DMA: CH0_AL2_WRITE_ADDR_TRIG, CH1_AL2_WRITE_ADDR_TRIG, ..., CH10_AL2_WRITE_ADDR_TRIG, CH11_AL2_WRITE_ADDR_TRIG 寄存器

偏移地址: 0x02c, 0x06c, ..., 0x2ac, 0x2ec

表132。
CH0_AL2_WRITE_ADD_R_TRIG,
CH1_AL2_WRITE_ADD_R_TRIG, ...
CH10_AL2_WRITE_AD_DR_TRIG,
CH11_AL2_WRITE_DR_TRIG 寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: CH0_AL3_CTRL, CH1_AL3_CTRL, ..., CH10_AL3_CTRL, CH11_AL3_CTRL 寄存器

偏移地址: 0x030, 0x070, ..., 0x2b0, 0x2f0

表133。
CH0_AL3_CTRL,
CH1_AL3_CTRL, ...
, CH10_AL3_CTRL
, CH11_AL3_CTRL
L 寄存器

位	描述	类型	复位值
31:0	通道 NCTRL 寄存器别名	读写	-

DMA: CH0_AL3_WRITE_ADDR, CH1_AL3_WRITE_ADDR, ..., CH10_AL3_WRITE_ADDR, CH11_AL3_WRITE_ADDR 寄存器

偏移量: 0x034, 0x074, ..., 0x2b4, 0x2f4

表134。
CH0_AL3_WRITE_ADD_R,
CH1_AL3_WRITE_ADD_R, ...
CH10_AL3_WRITE_AD_DR,
CH11_AL3_WRITE_DR 寄存器

位	描述	类型	复位值
31:0	通道 NWRITE_ADDR 寄存器别名	读写	-

DMA: CH0_AL3_TRANS_COUNT, CH1_AL3_TRANS_COUNT, ..., CH10_AL3_TRANS_COUNT, CH11_AL3_TRANS_COUNT 寄存器

偏移量: 0x038, 0x078, ..., 0x2b8, 0x2f8

表135。
CH0_AL3_TRANS_CO_NT,
CH1_AL3_TRANS_CO_NT, ...
CH10_AL3_TRANS_CO_NT,
CH11_AL3_TRANS_CO_NT 寄存器

位	描述	类型	复位值
31:0	通道 NTRANS_COUNT 寄存器别名	读写	-

DMA: CH0_AL3_READ_ADDR_TRIG, CH1_AL3_READ_ADDR_TRIG, ..., CH10_AL3_READ_ADDR_TRIG, CH11_AL3_READ_ADDR_TRIG 寄存器

偏移量: 0x03c, 0x07c, ..., 0x2bc, 0x2fc

表136。
CH0_AL3_READ_ADDR_TRIG,
CH1_AL3_READ_ADDR_TRIG, ...
CH10_AL3_READ_ADD_DR_TRIG,
CH11_AL3_READ_DR_TRIG 寄存器

位	描述	类型	复位值
31:0	通道 NREAD_ADDR 寄存器别名 这是一个触发寄存器 (0xc)。写入非零值将重新加载通道计数器并启动该通道。	读写	-

DMA: INTR 寄存器

偏移量: 0x400

描述

中断状态 (原始)

Table 137. INTR Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	<p>Raw interrupt status for DMA Channels 0..15. Bit n corresponds to channel n.</p> <p>Ignores any masking or forcing. Channel interrupts can be cleared by writing a bit mask to INTR, INTS0 or INTS1.</p> <p>Channel interrupts can be routed to either of two system-level IRQs based on INTE0 and INTE1.</p> <p>This can be used vector different channel interrupts to different ISRs: this might be done to allow NVIC IRQ preemption for more time-critical channels, or to spread IRQ load across different cores.</p> <p>It is also valid to ignore this behaviour and just use INTE0/INTS0/IRQ 0.</p>	WC	0x0000

DMA: INTE0 Register

Offset: 0x404

Description

Interrupt Enables for IRQ 0

Table 138. INTE0 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Set bit n to pass interrupts from channel n to DMA IRQ 0.	RW	0x0000

DMA: INTF0 Register

Offset: 0x408

Description

Force Interrupts

Table 139. INTF0 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Write 1s to force the corresponding bits in INTE0. The interrupt remains asserted until INTF0 is cleared.	RW	0x0000

DMA: INTS0 Register

Offset: 0x40c

Description

Interrupt Status for IRQ 0

表137。INTR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	DMA 通道 0 至 15 的原始中断状态。位 n 对应通道 n。 忽略任何屏蔽或强制操作。可通过向 INTR、INTS0 或 INTS1 写入位掩码以清除通道中断。 根据 INTE0 和 INTE1，通道中断可路由至两个系统级 IRQ 的任一。 此功能可用于将不同通道中断向量分配至不同中断服务程序（ISR）：此举可能旨在允许对更具时效性的通道执行 NVIC IRQ 抢占，或将 IRQ 负载分散至不同核心。 同样可忽略该行为，仅使用 INTE0/INTS0/IRQ 0。	WC	0x0000

DMA：INTE0 寄存器

偏移: 0x404

描述

IRQ 0 中断使能

表 138。INTE0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	设置位 n 以使通道 n 的中断传递至 DMA IRQ 0。	读写	0x0000

DMA：INTF0 寄存器

偏移: 0x408

描述

强制中断

表 139。INTF0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	写入 1 以强制 INTE0 中相应位的中断。中断将持续有效，直到 INTF0 被清除。	读写	0x0000

DMA：INTS0 寄存器

偏移: 0x40c

描述

IRQ 0 的中断状态

Table 140. INTS0 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Indicates active channel interrupt requests which are currently causing IRQ 0 to be asserted. Channel interrupts can be cleared by writing a bit mask here.	WC	0x0000

DMA: INTE1 Register

Offset: 0x414

Description

Interrupt Enables for IRQ 1

Table 141. INTE1 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Set bit n to pass interrupts from channel n to DMA IRQ 1.	RW	0x0000

DMA: INTF1 Register

Offset: 0x418

Description

Force Interrupts for IRQ 1

Table 142. INTF1 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Write 1s to force the corresponding bits in INTE0. The interrupt remains asserted until INTF0 is cleared.	RW	0x0000

DMA: INTS1 Register

Offset: 0x41c

Description

Interrupt Status (masked) for IRQ 1

Table 143. INTS1 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Indicates active channel interrupt requests which are currently causing IRQ 1 to be asserted. Channel interrupts can be cleared by writing a bit mask here.	WC	0x0000

DMA: TIMER0, TIMER1, TIMER2, TIMER3 Registers

Offsets: 0x420, 0x424, 0x428, 0x42c

Description

Pacing (X/Y) Fractional Timer

The pacing timer produces TREQ assertions at a rate set by $((X/Y) * \text{sys_clk})$. This equation is evaluated every sys_clk cycles and therefore can only generate TREQs at a rate of 1 per sys_clk (i.e. permanent TREQ) or less.

Table 144. TIMER0, TIMER1, TIMER2, TIMER3 Registers

Bits	Description	Type	Reset
31:16	X: Pacing Timer Dividend. Specifies the X value for the (X/Y) fractional timer.	RW	0x0000

表140. INTS0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	指示当前激活的通道中断请求，这些请求正导致 IRQ 0 被触发。 可通过向此处写入位掩码以清除通道中断。	WC	0x0000

DMA：INTE1 寄存器

偏移: 0x414

描述

IRQ 1 的中断使能

表141. INTE1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	设置第 n 位以将第 n 通道的中断传递至 DMA IRQ 1。	读写	0x0000

DMA：INTF1 寄存器

偏移: 0x418

描述

IRQ 1 的强制中断

表142. INTF1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	写入 1 以强制 INTFO 中相应位的中断。中断将持续有效，直到 INTFO 被清除。	读写	0x0000

DMA：INTS1 寄存器

偏移: 0x41c

描述

IRQ 1 的中断状态（屏蔽）

表143. INTS1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	指示当前激活的通道中断请求，这些请求正导致 IRQ 1 被触发。 可通过向此处写入位掩码以清除通道中断。	WC	0x0000

DMA：TIMER0、TIMER1、TIMER2、TIMER3 寄存器

偏移: 0x420, 0x424, 0x428, 0x42c

描述

节拍 (X/Y) 分数定时器

该节拍定时器以 $((X/Y) * \text{sys_clk})$ 设定的速率产生 TREQ 断言。该方程每个 sys_clk 周期计算一次，因此只能以每个 sys_clk 最多 1 次（即持续 TREQ）或更低速率产生 TREQ。

表 144。 TIMER0,
TIMER1, TIMER2,
TIMER3 寄存器

位	描述	类型	复位值
31:16	X：分频定时器被除数。指定 (X/Y) 分数定时器的 X 值。	读写	0x0000

Bits	Description	Type	Reset
15:0	Y: Pacing Timer Divisor. Specifies the Y value for the (X/Y) fractional timer.	RW	0x0000

DMA: MULTI_CHAN_TRIGGER Register

Offset: 0x430

Description

Trigger one or more channels simultaneously

Table 145.
MULTI_CHAN_TRIGGER Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Each bit in this register corresponds to a DMA channel. Writing a 1 to the relevant bit is the same as writing to that channel's trigger register; the channel will start if it is currently enabled and not already busy.	SC	0x0000

DMA: SNIFF_CTRL Register

Offset: 0x434

Description

Sniffer Control

Table 146.
SNIFF_CTRL Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	OUT_INV : If set, the result appears inverted (bitwise complement) when read. This does not affect the way the checksum is calculated; the result is transformed on-the-fly between the result register and the bus.	RW	0x0
10	OUT_REV : If set, the result appears bit-reversed when read. This does not affect the way the checksum is calculated; the result is transformed on-the-fly between the result register and the bus.	RW	0x0
9	BSWAP : Locally perform a byte reverse on the sniffed data, before feeding into checksum. Note that the sniff hardware is downstream of the DMA channel byteswap performed in the read master: if channel CTRL_BSWAP and SNIFF_CTRL_BSWAP are both enabled, their effects cancel from the sniffer's point of view.	RW	0x0
8:5	CALC Enumerated values: 0x0 → CRC32: Calculate a CRC-32 (IEEE802.3 polynomial)	RW	0x0
	0x1 → CRC32R: Calculate a CRC-32 (IEEE802.3 polynomial) with bit reversed data		
	0x2 → CRC16: Calculate a CRC-16-CCITT		
	0x3 → CRC16R: Calculate a CRC-16-CCITT with bit reversed data		
	0xe → EVEN: XOR reduction over all data. == 1 if the total 1 population count is odd.		
	0xf → SUM: Calculate a simple 32-bit checksum (addition with a 32 bit accumulator)		

位	描述	类型	复位值
15:0	Y：分频定时器除数。指定 (X/Y) 分数定时器的 Y 值。	读写	0x0000

DMA: MULTI_CHAN_TRIGGER 寄存器

偏移: 0x430

描述

同时触发一个或多个通道

表 145。
MULTI_CHAN_TRIGGER
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	该寄存器的每个位对应一个 DMA 通道。向相关位写入 1 等同于写入该通道的触发寄存器；若通道已启用且当前未忙，则将启动该通道。	SC	0x0000

DMA: SNIFF_CTRL 寄存器

偏移: 0x434

描述

嗅探器控制

表 146。
SNIFF_CTRL 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	OUT_INV : 若设置，读取时结果将呈现取反（按位补码）。此设置不影响校验和的计算方式；结果在结果寄存器与总线之间实时转换。	读写	0x0
10	OUT_REV : 若设置，读取时结果按位反转显示。此设置不影响校验和的计算方式；结果在结果寄存器与总线之间实时转换。	读写	0x0
9	BSWAP : 在输入校验和前，对嗅探数据局部执行字节反转。 请注意，嗅探硬件位于 DMA 通道字节交换操作之后的下游：若通道 CTRL_BSWAP 与 SNIFF_CTRL_BSWAP 均启用，则从嗅探器视角两者效果相互抵消。	读写	0x0
8:5	CALC 枚举值： 0x0 → CRC32: 计算 CRC-32 (IEEE802.3 多项式)	读写	0x0
	0x1 → CRC32R: 计算位反转数据的 CRC-32 (IEEE802.3 多项式)		
	0x2 → CRC16: 计算 CRC-16-CCITT		
	0x3 → CRC16R: 计算位反转数据的 CRC-16-CCITT		
	0xe → EVEN: 对所有数据执行 XOR 归约。== 1 若总计1的人口数为奇数。		
	0xf → SUM: 计算简单的32位校验和（使用32位累加器相加）		

Bits	Description	Type	Reset
4:1	DMACH : DMA channel for Sniffer to observe	RW	0x0
0	EN : Enable sniffer	RW	0x0

DMA: SNIFF_DATA Register

Offset: 0x438

Description

Data accumulator for sniff hardware

Table 147.
SNIFF_DATA Register

Bits	Description	Type	Reset
31:0	Write an initial seed value here before starting a DMA transfer on the channel indicated by SNIFF_CTRL_DMACH. The hardware will update this register each time it observes a read from the indicated channel. Once the channel completes, the final result can be read from this register.	RW	0x00000000

DMA: FIFO_LEVELS Register

Offset: 0x440

Description

Debug RAF, WAF, TDF levels

Table 148.
FIFO_LEVELS Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:16	RAF_LVL : Current Read-Address-FIFO fill level	RO	0x00
15:8	WAF_LVL : Current Write-Address-FIFO fill level	RO	0x00
7:0	TDF_LVL : Current Transfer-Data-FIFO fill level	RO	0x00

DMA: CHAN_ABORT Register

Offset: 0x444

Description

Abort an in-progress transfer sequence on one or more channels

Table 149.
CHAN_ABORT
Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Each bit corresponds to a channel. Writing a 1 aborts whatever transfer sequence is in progress on that channel. The bit will remain high until any in-flight transfers have been flushed through the address and data FIFOs. After writing, this register must be polled until it returns all-zero. Until this point, it is unsafe to restart the channel.	SC	0x0000

DMA: N_CHANNELS Register

Offset: 0x448

Table 150.
N_CHANNELS Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-

位	描述	类型	复位值
4:1	DMACH : 供嗅探器观察的DMA通道	读写	0x0
0	EN : 启用嗅探器	读写	0x0

DMA: SNIFF_DATA 寄存器

偏移量: 0x438

说明

嗅探硬件数据累加器

表147
SNIFF_DATA 寄存器

位	描述	类型	复位值
31:0	在开始对由 SNIFF_CTRL_DMACH 指示的通道进行DMA传输前, 请在此写入初始种子值。硬件将在每次侦测到对该通道的读取时更新此寄存器。通道完成后, 可从该寄存器读取最终结果。	读写	0x00000000

DMA: FIFO_LEVELS 寄存器

偏移量: 0x440

说明

调试 RAF、WAF、TDF 级别

表148
FIFO_LEVELS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	RAF_LVL : 当前读地址FIFO填充级别	只读	0x00
15:8	WAF_LVL : 当前写地址FIFO填充级别	只读	0x00
7:0	TDF_LVL : 当前传输数据FIFO填充级别	只读	0x00

DMA: CHAN_ABORT 寄存器

偏移: 0x444

描述

中止一个或多个通道上的正在进行传输序列

表149。
CHAN_ABORT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	每个位对应一个通道。写入1会中止该通道上正在进行的任何传输序列。该位将保持高电平, 直到所有在途传输通过地址和数据FIFO完全刷新。 写入后, 必须轮询该寄存器, 直到其返回全零。在此之前, 重启通道是不安全的。	SC	0x0000

DMA: N_CHANNELS 寄存器

偏移: 0x448

表150。
N_CHANNELS 寄存器

位	描述	类型	复位值
31:5	保留。	-	-

Bits	Description	Type	Reset
4:0	The number of channels this DMA instance is equipped with. This DMA supports up to 16 hardware channels, but can be configured with as few as one, to minimise silicon area.	RO	-

DMA: CH0_DBG_CTDREQ, CH1_DBG_CTDREQ, ..., CH10_DBG_CTDREQ, CH11_DBG_CTDREQ Registers

Offsets: 0x800, 0x840, ..., 0xa80, 0xac0

Table 151.
CH0_DBG_CTDREQ,
CH1_DBG_CTDREQ, ...,
CH10_DBG_CTDREQ,
CH11_DBG_CTDREQ
Registers

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	Read: get channel DREQ counter (i.e. how many accesses the DMA expects it can perform on the peripheral without overflow/underflow. Write any value: clears the counter, and cause channel to re-initiate DREQ handshake.	WC	0x00

DMA: CH0_DBG_TCR, CH1_DBG_TCR, ..., CH10_DBG_TCR, CH11_DBG_TCR Registers

Offsets: 0x804, 0x844, ..., 0xa84, 0xac4

Table 152.
CH0_DBG_TCR,
CH1_DBG_TCR, ...,
CH10_DBG_TCR,
CH11_DBG_TCR
Registers

Bits	Description	Type	Reset
31:0	Read to get channel TRANS_COUNT reload value, i.e. the length of the next transfer	RO	0x00000000

2.6. Memory

RP2040 has embedded ROM and SRAM, and access to external Flash via a QSPI interface. Details of internal memory are given below.

2.6.1. ROM

A 16kB read-only memory (ROM) is at address `0x00000000`. The ROM contents are fixed at the time the silicon is manufactured. It contains:

- Initial startup routine
- Flash boot sequence
- Flash programming routines
- USB mass storage device with UF2 support
- Utility libraries such as fast floating point

The boot sequence of the chip is defined in [Section 2.8.1](#), and the ROM contents is described in more detail in [Section 2.8](#). The full source code for the RP2040 bootrom is available at:

[pico-bootrom](#)

The ROM offers single-cycle read-only bus access, and is on a dedicated AHB-Lite arbiter, so it can be accessed simultaneously with other memory devices. Attempting to write to the ROM has no effect (no bus fault is generated).

位	描述	类型	复位值
4:0	本DMA实例所配置的通道数量。该DMA最多支持16个硬件通道，但可配置为最少1个，以减少芯片面积。	只读	-

DMA: CH0_DBG_CTDREQ, CH1_DBG_CTDREQ, ..., CH10_DBG_CTDREQ, CH11_DBG_CTDREQ 寄存器

偏移: 0x800, 0x840, ..., 0xa80, 0xac0

表151。
CH0_DBG_CTDREQ,
, CH1_DBG_CTDREQ
, ..., CH10_DBG_C
TDREQ, CH11_DB
G_CTDREQ 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	读取: 获取通道DREQ计数器（即DMA预计可对外设执行而不会溢出或欠载的访问次数）。写入任意值: 清除计数器，并使通道重新启动DREQ握手。	WC	0x00

DMA: CH0_DBG_TCR, CH1_DBG_TCR, ..., CH10_DBG_TCR, CH11_DBG_TCR 寄存器

偏移量: 0x804, 0x844, ..., 0xa84, 0xac4

表152。
CH0_DBG_TCR,
CH1_DBG_TCR, ...
CH10_DBG_TCR,
CH11_DBG_TCR
寄存器

位	描述	类型	复位值
31:0	读取通道 TRANS_COUNT 重新加载值，即下一次传输的长度	只读	0x00000000

2.6. 内存

RP2040内置ROM和SRAM，并通过QSPI接口访问外部Flash。内部存储器详细信息如下。

2.6.1. 只读存储器 (ROM)

一块16kB只读存储器 (ROM) 位于地址 **0x00000000**. ROM内容在硅片制造时固定，包含：

- 初始启动程序
- Flash 启动序列
- Flash 编程例程
- 支持 UF2 的 USB 大容量存储设备
- 快速浮点等实用库

芯片的启动序列定义于第 2.8.1 节，ROM 内容在第 2.8 节中有更详细的描述。RP2040 启动 ROM 的完整源代码可在以下位置获取：

[pico-bootrom](#)

ROM 提供单周期只读总线访问，且位于专用的 AHB-Lite 仲裁器上，因此可与其他存储设备同时访问。尝试写入 ROM 不会产生任何效果（不会触发总线错误）。

2.6.2. SRAM

There is a total of 264kB of on-chip SRAM. Physically this is partitioned into six banks, as this vastly improves memory bandwidth for multiple masters, but software may treat it as a single 264kB memory region. There are no restrictions on what is stored in each bank: processor code, data buffers, or a mixture. There are four 16k x 32-bit banks (64kB each) and two 1k x 32-bit banks (4kB each).

⚠️ IMPORTANT

Banking is a *physical* partitioning of SRAM which improves performance by allowing multiple simultaneous accesses. *Logically* there is a single 264kB contiguous memory.

Each SRAM bank is accessed via a dedicated AHB-Lite arbiter. This means different bus masters can access different SRAM banks in parallel, so up to four 32-bit SRAM accesses can take place every system clock cycle (one per master).

SRAM is mapped to system addresses starting at `0x20000000`. The first 256kB address region is word-striped across the four larger banks, which provides a significant memory parallelism benefits for most use cases.

Consecutive words in the system address space are routed to different RAM banks as shown in [Table 153](#).

Table 153. SRAM bank0/1/2/3 striped mapping.

System address	SRAM Bank	SRAM word address
<code>0x20000000</code>	Bank 0	0
<code>0x20000004</code>	Bank 1	0
<code>0x20000008</code>	Bank 2	0
<code>0x2000000c</code>	Bank 3	0
<code>0x20000010</code>	Bank 0	1
<code>0x20000014</code>	Bank 1	1
<code>0x20000018</code>	Bank 2	1
<code>0x2000001c</code>	Bank 3	1
<code>0x20000020</code>	Bank 0	2
<code>0x20000024</code>	Bank 1	2
<code>0x20000028</code>	Bank 2	2
<code>0x2000002c</code>	Bank 3	2
etc		

The next two 4kB regions (starting at `0x20040000` and `0x20041000`) are mapped directly to the smaller, 4kB memory banks. Software *may* choose to use these for per-core purposes, e.g. stack and frequently-executed code, guaranteeing that the processors never stall on these accesses. However, like all SRAM on RP2040, these banks have single-cycle access from *all* masters providing no other masters are accessing the bank in the same cycle, so it is reasonable to treat memory as a single 264kB device.

The four 64kB banks are also available at a non-striped mirror. The four 64kB regions starting at `0x21000000`, `0x21010000`, `0x21020000`, `0x21030000` are each mapped directly to one of the four 64kB SRAM banks. Software can explicitly allocate data and code across the physical memory banks, for improved memory performance in exceptionally demanding cases. This is often unnecessary, as memory striping usually provides sufficient parallelism with less software complexity.

The non-striped mirror starts at an offset of +16MB above the base of SRAM, as this is the maximum offset that allows ARMv6M subroutine calls between the smaller banks and the non-striped larger banks.

2.6.2. 静态随机存取存储器 (SRAM)

芯片内共有 264kB SRAM。物理上划分为六个存储块，这显著提升了多主设备的内存带宽，但软件可将其视为单一 264kB 内存区域。对每个存储块中的存储内容无限制：可为处理器代码、数据缓冲区或二者混合。共有四个16k x 32位存储区（每个64kB）和两个1k x 32位存储区（每个4kB）。

重要

分区是一种物理划分SRAM的方法，通过允许多个同时访问来提升性能。逻辑上存在一个单一的264kB连续内存空间。

每个SRAM存储区均通过专用的AHB-Lite仲裁器进行访问。这意味着不同的总线主控可以并行访问不同的SRAM存储区，因此每个系统时钟周期最多可执行四次32位SRAM访问（每个主控一次）。

SRAM映射至系统地址，起始地址为 `0x20000000`。首个256kB地址区域采用字条带方式跨越四个较大存储区分布，为大多数应用提供显著的内存并行优势。

系统地址空间中连续的字按照表153所示路由至不同的RAM存储区。

表153。SRAM
存储区0/1/2/3条带
映射示意。

系统地址	SRAM银行	SRAM字地址
<code>0x20000000</code>	银行0	0
<code>0x20000004</code>	银行1	0
<code>0x20000008</code>	银行2	0
<code>0x2000000c</code>	银行3	0
<code>0x20000010</code>	银行0	1
<code>0x20000014</code>	银行1	1
<code>0x20000018</code>	银行2	1
<code>0x2000001c</code>	银行3	1
<code>0x20000020</code>	银行0	2
<code>0x20000024</code>	银行1	2
<code>0x20000028</code>	银行2	2
<code>0x2000002c</code>	银行3	2
等等		

接下来的两个4kB区域（起始地址为 `0x20040000` 和 `0x20041000`）直接映射到较小的4kB内存块。

软件可以选择将这些用于每核用途，例如堆栈和频繁执行的代码，以确保处理器访问这些区域时不会发生停滞。然而，与 RP2040 上所有 SRAM 一样，这些内存块对所有主控器提供单周期访问，前提是同一周期内没有其他主控器访问该内存块，因此将内存视为单一的264kB设备是合理的。

这四个64kB的内存块也可通过非条带化镜像访问。从 `0x21000000`、`0x21010000`、`0x21020000`、`0x21030000`起始的四个64kB区域各自直接映射到四个64kB SRAM内存块之一。软件可以明确分配数据和代码至物理内存块，以在极端要求情况下提升内存性能。这通常是不必要的，因为内存条带通常能够提供足够的并行性，并且软件复杂度更低。

非条带镜像起始于SRAM基址上方偏移+16MB处，这是允许ARMv6M在较小存储区与非条带较大存储区之间进行子程序调用的最大偏移。

2.6.2.1. Other On-chip Memory

Besides the 264kB main memory, there are two other dedicated RAM blocks that may be used in some circumstances:

- If flash XIP caching is disabled, the cache becomes available as a 16kB memory starting at `0x15000000`
- If the USB is not used, the USB data DPRAM can be used as a 4kB memory starting at `0x50100000`

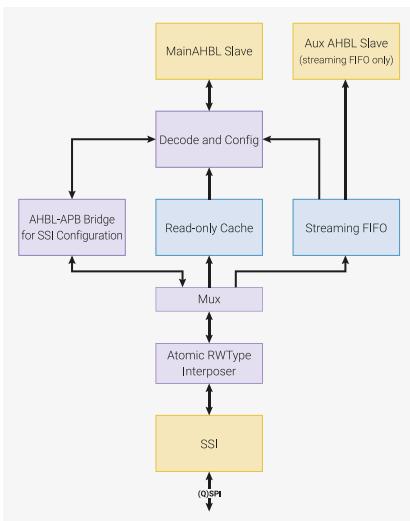
This gives a total of 284kB of on-chip SRAM. There are no restrictions on how these memories are used, e.g. it is possible to execute code from the USB data RAM if you choose.

2.6.3. Flash

External Flash is accessed via the QSPI interface using the execute-in-place (XIP) hardware. This allows an external flash memory to be addressed and accessed by the system as though it were internal memory. Bus reads to a 16MB memory window starting at `0x10000000` are translated into a serial flash transfer, and the result is returned to the master that initiated the read. This process is transparent to the master, so a processor can execute code from the external flash without first copying the code to internal memory, hence "execute in place". An internal cache remembers the contents of recently-accessed flash locations, which accelerates the average bandwidth and latency of the interface.

Once correctly configured by RP2040's bootrom and the flash second stage, the XIP hardware is largely transparent, and software can treat flash as a large read-only memory. However, it does provide a number of additional features to serve more demanding software use cases.

Figure 14. Flash execute-in-place (XIP) subsystem. System accesses via the main AHB-Lite slave are decoded to determine if they are XIP accesses, direct accesses to the SSI e.g. for configuration, or accesses to various other hardware and control registers in the XIP subsystem. XIP accesses are first looked up in the cache, to accelerate accesses to recently-used data. If the data is not found in the cache, an external serial access is generated via the SSI, and the resulting data is stored in the cache and forwarded on to the system bus.



NOTE

The serial flash interface is configured by the flash second stage when using the SDK to run at an integer divider of the system clock. All the included second stage boot implementations support a `PICO_FLASH_SPI_CLKDIV` setting (e.g. defaulted to 4 in https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/boot_stage2/boot2_w25q080.S to make the default interface speed $125/4 = 31.25\text{MHz}$). This divider can be overridden by specifying `PICO_FLASH_SPI_CLKDIV` in the particular board config header used with the SDK.

2.6.3.1. XIP Cache

The cache is 16kB, two way set-associative, 1 cycle hit. It is internal to the XIP subsystem, and only affects accesses to XIP flash, so software does not have to consider cache coherence, unless performing flash programming operations. It caches reads from a 24-bit flash address space, which is mirrored multiple times in the RP2040 address space, each alias having different caching behaviour. The eight MSBs of the system address are used for segment decode, leaving 24 bits for flash addressing, so the maximum supported flash size (for XIP operation) is 16MB. The available mirrors

2.6.2.1. 其他片上存储器

除了264kB主存储器外，还有两个在某些情况下可用的专用RAM块：

- 如果禁用闪存XIP缓存，该缓存将作为一个16kB的存储器，可从 `0x15000000` 开始使用。
- 如果未使用USB，USB数据DPRAM可作为一个4kB存储器，从 `0x50100000` 开始使用。

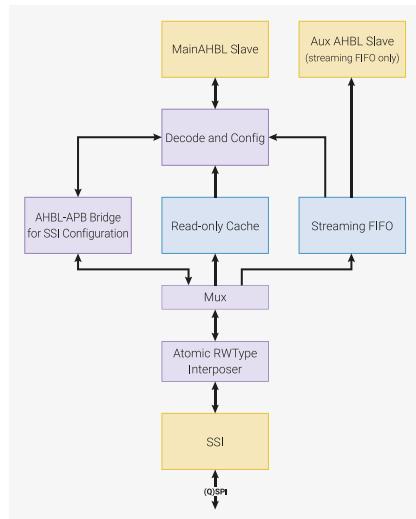
这样，片上SRAM总容量达到284kB。对于这些存储器的使用没有限制，例如，可以选择从USB数据RAM执行代码。

2.6.3. 闪存

外部闪存通过QSPI接口使用执行就地（XIP）硬件进行访问。这使得系统能够将外部闪存作为内部内存一样进行寻址和访问。对起始地址为 `0x1` 的16MB内存窗口的总线读取 `00000000` 会被转换为串行闪存传输，结果返回给发起读取的主设备。该过程对主设备透明，因此处理器可以直接从外部闪存执行代码，无需先将代码复制到内部内存，故称“执行就地”。内部缓存会记录最近访问的闪存内容，从而提升接口的平均带宽和延迟表现。

一旦由RP2040引导ROM及闪存第二阶段正确配置，XIP硬件基本透明，软件可将闪存视为大型只读存储器。然而，它确实提供多项附加功能，以满足更复杂的软件应用需求。

图14。Flash
执行就地（XIP）子
系统。系统通过
主 AHB-Lite 从属设
备的访问会被解
码，以确认访问属
于 XIP 访问
、对 SSI 的直
接访问（例如用
于配置），或对 XI
P 子系统中其他各
种硬件与控制寄
存器的访问。XIP 访
问首先在缓存中
查询，以加速
对最近使用数
据的访问。若缓
存中无数据，则通
过 SSI 发起外部串
行访问，随后
将获取的数据存
入缓存，并
转发至系统总线。
转自：https://www.raspberrypi.org/documentation/hardware/rpi4/technical/flash-xip-subsystem.pdf



注意

使用SDK时，闪存第二阶段将串行闪存接口配置为系统时钟的整数分频。所有包含的第二阶段引导实现均支持 `PICO_FLASH_SPI_CLKDIV` 设置（例如，在 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/boot_stage2/boot2_w25q080.S 中默认值为 4，使默认接口速度为 $125/4 = 31.25\text{MHz}$ ）。该分频设置可通过在 SDK 所用特定板配置头文件中指定 `PICO_FLASH_SPI_CLKDIV` 进行覆盖。

2.6.3.1. XIP 缓存

缓存容量为16kB，双路组相联，命中时间为1周期。它属于XIP子系统内部，仅影响对XIP闪存的访问，因此软件无需考虑缓存一致性，除非执行闪存编程操作。它缓存来自24位闪存地址空间的读取，该地址空间在RP2040地址空间中被多次镜像，每个镜像具有不同的缓存行为。系统地址的最高八位用于段解码，剩余24位用于闪存寻址，因此XIP操作支持的最大闪存容量为16MB。可用的镜像地址

are:

- **0x10…** XIP access, cacheable, allocating - Normal cache operation
- **0x11…** XIP access, cacheable, non-allocating - Check for hit, don't update cache on miss
- **0x12…** XIP access, non-cacheable, allocating - Don't check for hit, always update cache
- **0x13…** XIP access, non-cacheable, non-allocating - Bypass cache completely
- **0x15…** Use XIP cache as SRAM bank, mirrored across entire segment

If the cache is disabled, via the **CTRL.EN** register bit, then all four of the XIP aliases (**0x10** to **0x13**) will bypass the cache, and access the flash directly. This has a significant impact on XIP code execution performance.

Access to the **0x15…** segment produces a bus error unless the cache is disabled by clearing **CTRL.EN**. Once the cache is disabled, this region behaves as an additional 16kB SRAM bank. Reads and writes are one cycle, but there is a wait state on consecutive write-read sequences, i.e. there is no write forwarding buffer.

2.6.3.2. Cache Flushing and Maintenance

The **FLUSH** register allows the entire cache contents to be flushed. This is necessary if software has reprogrammed the flash contents, and needs to clear out stale data and code, without performing a reboot. Cache flushes are triggered either manually by writing 1 to **FLUSH**, or automatically when the XIP block is brought out of reset. The flush is implemented by zeroing the cache tag memory using an internal counter, which takes just over 1024 clock cycles (16kB total size / 8 bytes per line / 2 ways per set).

Flushing the cache whilst accessing flash data (perhaps initiating the flush on one core whilst another core may be executing code from flash) is a safe operation, but any master accessing flash data while the flush is in progress will be stalled until completion.

⚠ CAUTION

The cache-as-SRAM alias (**0x15…**) must not be written whilst a cache flush is in progress. Before writing for the first time, if a cache flush has recently been initiated (e.g. via a watchdog reset), a dummy read from **FLUSH** is recommended to ensure the cache flush has completed. Writing to cache-as-SRAM whilst a flush is in progress can corrupt the data memory contents.

A complete cache flush dramatically slows subsequent code execution, until the cache "warms up" again. There is an alternative, which allows cache contents corresponding to only a certain address range to be invalidated. A write to the **0x10…** mirror will look up the addressed location in the cache, and delete any matching entry found. Writing to all word-aligned locations in an address range (e.g. a flash sector that has just been erased and reprogrammed) therefore eliminates the possibility of stale cached data in this range, without suffering the effects of a complete cache flush.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/flash/cache_perfctr/flash_cache_perfctr.c Lines 30 - 55

```

30     // Flush cache to make sure we miss the first time we access test_data
31     xip_ctrl_hw->flush = 1;
32     while (!(xip_ctrl_hw->stat & XIP_STAT_FLUSH_READY_BITS))
33         tight_loop_contents();
34
35     // Clear counters (write any value to clear)
36     xip_ctrl_hw->ctr_acc = 1;
37     xip_ctrl_hw->ctr_hit = 1;
38
39     (void) *test_data_ptr;
40     check(xip_ctrl_hw->ctr_hit == 0 && xip_ctrl_hw->ctr_acc == 1,
41           "First access to data should miss");
42
43     (void) *test_data_ptr;
44     check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 2,

```

为：

- **0x10…** XIP访问，可缓存，分配——正常缓存操作
- **0x11…** XIP访问，可缓存，不分配——检查命中，未命中时不更新缓存
- **0x12…** XIP访问，不缓存，分配——不检查命中，始终更新缓存
- **0x13…** XIP访问，非缓存，非分配——完全绕过缓存
- **0x15…** 使用XIP缓存作为SRAM组，镜像映射至整个段

如果通过CTRL.EN寄存器位禁用缓存，则所有四个XIP别名（**0x10**至**0x13**）将绕过缓存，直接访问闪存。这对XIP代码执行性能具有显著影响。

除非通过清除CTRL.EN禁用缓存，否则访问**0x15…**段会产生总线错误。一旦禁用缓存，该区域表现为额外的16KB SRAM组。读写操作均为一个周期，但在连续写-读序列中存在等待周期，即不存在写转发缓冲。

2.6.3.2. 缓存刷新与维护

FLUSH寄存器允许刷新整个缓存内容。如果软件重新编程了闪存内容，需要在不重启的情况下清除过期数据和代码，此操作是必要的。缓存刷新可通过向FLUSH写入1手动触发，或在将XIP块从复位状态切出时自动触发。刷新操作通过使用内部计数器将缓存标签存储器清零实现，耗时略超过1024个时钟周期（16kB总容量 / 每行8字节 / 每组2路）。

在访问闪存数据时执行缓存刷新（例如一核发起刷新，另一核可能正在执行闪存代码）是安全的，但任何在刷新进行期间访问闪存数据的主控设备将被阻塞，直至刷新完成。

⚠ 注意

缓存刷新进行时，禁止写入缓存映射的SRAM别名区域（**0x15…**）。首次写入前，如近期已启动缓存刷新（例如通过看门狗复位），建议对FLUSH执行一次虚假读取，以确保缓存刷新完成。刷新进行时写入缓存映射为SRAM可能会破坏数据存储内容。

完全缓存刷新会显著降低后续代码的执行速度，直至缓存重新“预热”。存在另一种方法，允许仅使对应于特定地址范围的缓存内容失效。对**0x10…**镜像的写操作将查找缓存中的对应地址位置，并删除任何匹配的条目。因此，对地址范围内所有字对齐位置的写入操作（例如刚擦除并重新编程的闪存扇区）可消除该范围内陈旧的缓存数据，且不会受到完全缓存刷新带来的影响。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/cache_perfctr/flash_cache_perfctr.c 第30至55行

```

30     // 刷新缓存，确保首次访问 test_data 时缓存失效
31     xip_ctrl_hw->flush = 1;
32     while (!(xip_ctrl_hw->stat & XIP_STAT_FLUSH_READY_BITS))
33         tight_loop_contents();
34
35     // 清除计数器（写入任意值以清零）
36     xip_ctrl_hw->ctr_acc = 1;
37     xip_ctrl_hw->ctr_hit = 1;
38
39     (void) *test_data_ptr;
40     check(xip_ctrl_hw->ctr_hit == 0 && xip_ctrl_hw->ctr_acc == 1,
41           "首次访问数据应未命中");
42
43     (void) *test_data_ptr;
44     check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 2,

```

```

45     "Second access to data should hit");
46
47     // Write to invalidate individual cache lines (64 bits)
48     // Writes must be directed to the cacheable, allocatable alias (address 0x10..._...)
49     *test_data_ptr = 0;
50     (void) *test_data_ptr;
51     check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 3,
52           "Should miss after invalidation");
53     (void) *test_data_ptr;
54     check(xip_ctrl_hw->ctr_hit == 2 && xip_ctrl_hw->ctr_acc == 4,
55           "Second access after invalidation should hit again");

```

2.6.3.3. SSI

The execute-in-place functionality is provided by the SSI interface, documented in [Section 4.10](#). It supports 1, 2 or 4-bit SPI flash interfaces (SPI, DSPI and QSPI), and can insert either an instruction prefix or mode continuation bits on each XIP access. This includes the possibility of issuing a standard `03h` serial flash read command for each access, allowing virtually any serial flash device to be used. The maximum SPI clock frequency is half the system clock frequency.

The SSI can also be used as a standard FIFO-based SPI master, with DMA support. This mode is used by the bootrom to extract the second stage bootloader from external flash (see [Section 2.8.1](#)). The bus interposer allows an atomic set, clear or XOR operation to be posted to SSI control registers, in the same manner as other memory-mapped IO on RP2040. This is described in more detail in [Section 2.1.2](#).

2.6.3.4. Flash Streaming and Auxiliary Bus Slave

As the flash is generally much larger than SRAM, it's often useful to stream chunks of data into memory from flash. It's convenient to have the DMA stream this data in the background while software in the foreground is doing other things, and it's even more convenient if code can continue to execute from flash whilst this takes place.

This doesn't interact well with standard XIP operation, because of the lengthy bus stalls forced on the DMA whilst the SSI is performing serial transfers. These stalls are tolerable for a processor, because an in-order processor tends to have nothing better to do while waiting for an instruction fetch to retire, and because typical code execution tends to have much higher cache hit rates than bulk streaming of infrequently accessed data. In contrast, stalling the DMA prevents any other active DMA channels from making progress during this time, which slows overall DMA throughput.

The `STREAM_ADDR` and `STREAM_CTR` registers are used to program a linear sequence of flash reads, which the XIP subsystem will perform in the background in a best-effort fashion. To minimise impact on code being executed from flash whilst the stream is ongoing, the streaming hardware has lower priority access to the SSI than regular XIP accesses, and there is a brief cooldown (seven cycles) between the last XIP cache miss and resuming streaming. This helps to avoid increase in initial access latency on XIP cache miss.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c Lines 45 - 48

```

45     while (!(xip_ctrl_hw->stat & XIP_STAT_FIFO_EMPTY))
46         (void) xip_ctrl_hw->stream_fifo;
47     xip_ctrl_hw->stream_addr = (uint32_t) &random_test_data[0];
48     xip_ctrl_hw->stream_ctr = count_of(random_test_data);

```

The streamed data is pushed to a small FIFO, which generates DREQ signals, telling the DMA to collect the streamed data. As the DMA does not initiate a read until after the data has been read from flash, the DMA is not stalled when accessing the data.

Although this scheme ensures that the data is ready in the streaming FIFO once the DREQ is asserted, the DMA can still be stalled if another master is currently stalled on the XIP slave, e.g. due to a cache miss. This is solved by the auxiliary bus slave, which is a simple bus interface providing access only to the streaming FIFO. This slave is exposed on the

```

45     "第二次访问数据命中");
46
47 // 写入以失效单个缓存行 (64 位)
48 // 写入必须指向可缓存且可分配的别名 (地址范围0x10.....)
49 *test_data_ptr = 0;
50 (void) *test_data_ptr;
51 check(xip_ctrl_hw->ctr_hit == 1 && xip_ctrl_hw->ctr_acc == 3,
52       "失效后应当丢失");
53 (void) *test_data_ptr;
54 check(xip_ctrl_hw->ctr_hit == 2 && xip_ctrl_hw->ctr_acc == 4,
55       "失效后第二次访问应再次命中");

```

2.6.3.3. SSI

执行就地 (execute-in-place) 功能由SSI接口提供，详见第4.10节。该接口支持1位、2位或4位SPI闪存接口（SPI、DSPI及QSPI），并可在每次XIP访问时插入指令前缀或模式续定位。这包括在每次访问时发出标准的 03h串行闪存读命令，几乎可支持任何串行闪存设备。最大SPI时钟频率为系统时钟频率的一半。

SSI还可作为标准的基于FIFO的SPI主控设备使用，并支持DMA。此模式被bootrom用于从外部闪存提取二级引导加载程序（见第2.8.1节）。总线中介器允许对SSI控制寄存器以原子方式执行置位、清除或异或操作，方式与RP2040上的其他内存映射IO相同。此操作详述于第2.1.2节。

2.6.3.4. 闪存流传输与辅助总线从属设备

由于闪存容量通常远大于SRAM，将数据块从闪存流式传输到内存中通常非常有用。在前台软件执行其他任务的同时，让DMA在后台流式传输这些数据，非常便利，更便利的是，在此过程中代码可以继续从闪存执行。

这与标准的XIP操作不兼容，因为在SSI执行串行传输时，DMA会被迫经历长时间的总线阻塞。这些阻塞对处理器而言是可容忍的，因为顺序处理器在等待指令取出完成期间通常无更优任务可做，且典型代码执行的缓存命中率显著高于对不常访问数据的大块流式传输。相反，阻塞DMA会阻止任何其他活动的DMA通道在此期间取得进展，从而降低整体DMA吞吐性能。

STREAM_ADDR和**STREAM_CTR**寄存器用于编程一系列线性闪存读取操作，XIP子系统将以尽最大努力的方式在后台执行这些操作。为了在流媒体持续传输期间最大限度地减少对从闪存执行的代码的影响，流媒体硬件对SSI的访问优先级低于常规的XIP访问，并且在最后一次XIP缓存未命中与恢复流传输之间存在短暂的冷却时间（七个周期）。此举有助于避免在XIP缓存未命中时初始访问延迟的增加。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c 第45至48行

```

45     while (!(xip_ctrl_hw->stat & XIP_STAT_FIFO_EMPTY))
46         (void) xip_ctrl_hw->stream_fifo;
47     xip_ctrl_hw->stream_addr = (uint32_t) &random_test_data[0];
48     xip_ctrl_hw->stream_ctr = count_of(random_test_data);

```

流式数据被推送至一个小型FIFO，该FIFO生成DREQ信号，以告知DMA收集流式数据。由于DMA只有在从闪存读取数据之后才会发起读取请求，因此在访问数据时，DMA不会发生阻塞。

尽管该方案确保一旦DREQ断言，数据即已准备好并位于流FIFO中，但如果其他主设备因XIP从设备阻塞（例如缓存未命中）而暂时停滞，DMA仍可能被阻塞。此问题由辅助总线从设备解决，该从设备为简单总线接口，仅提供对流FIFO的访问。此从设备公开于

FASTPERI arbiter, which services only native AHB-Lite peripherals which don't generate wait states, so the DMA will never experience stalls when accessing the FIFO at this address, assuming it has high bus priority.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c Lines 58 - 70

```

58     const uint dma_chan = 0;
59     dma_channel_config cfg = dma_channel_get_default_config(dma_chan);
60     channel_config_set_read_increment(&cfg, false);
61     channel_config_set_write_increment(&cfg, true);
62     channel_config_set_dreq(&cfg, DREQ_XIP_STREAM);
63     dma_channel_configure(
64         dma_chan,
65         &cfg,
66         (void *) buf, // Write addr
67         (const void *) XIP_AUX_BASE, // Read addr
68         count_of(random_test_data), // Transfer count
69         true // Start immediately!
70     );

```

2.6.3.5. Performance Counters

The XIP subsystem provides two performance counters. These are 32 bits in size, saturate upon reaching `0xffffffff`, and are cleared by writing any value. They count:

1. The total number of XIP accesses, to any alias
2. The number of XIP accesses which resulted in a cache hit

For common use cases, this allows the cache hit rate to be profiled.

2.6.3.6. List of XIP Registers

The XIP registers start at a base address of `0x14000000` (defined as `XIP_CTRL_BASE` in SDK).

Table 154. List of XIP registers

Offset	Name	Info
0x00	<code>CTRL</code>	Cache control
0x04	<code>FLUSH</code>	Cache Flush control
0x08	<code>STAT</code>	Cache Status
0x0c	<code>CTR_HIT</code>	Cache Hit counter
0x10	<code>CTR_ACC</code>	Cache Access counter
0x14	<code>STREAM_ADDR</code>	FIFO stream address
0x18	<code>STREAM_CTR</code>	FIFO stream control
0x1c	<code>STREAM_FIFO</code>	FIFO stream data

XIP: CTRL Register

Offset: 0x00

Description

Cache control

FASTPERI仲裁器，该仲裁器仅为不产生等待状态的本地AHB-Lite外设提供服务，因此在假定DMA拥有较高总线优先级的情况下，访问该地址处的FIFO时，DMA将不会发生阻塞。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/flash/xip_stream/flash_xip_stream.c 第58至70行

```

58     const uint dma_chan = 0;
59     dma_channel_config cfg = dma_channel_get_default_config(dma_chan);
60     channel_config_set_read_increment(&cfg, false);
61     channel_config_set_write_increment(&cfg, true);
62     channel_config_set_dreq(&cfg, DREQ_XIP_STREAM);
63     dma_channel_configure(
64         dma_chan,
65         &cfg,
66         (void *) buf,           // 写入地址
67         (const void *) XIP_AUX_BASE, // 读取地址
68         count_of(random_test_data), // 传输计数
69         true                   // 立即启动!
70     );

```

2.6.3.5. 性能计数器

XIP子系统提供两个性能计数器。两个计数器均为32位，达到饱和值 `0xffffffff` 时饱和，写入任意值可清零。计数项包括：

1. 任意别名的XIP访问总数
2. 因缓存命中而产生的XIP访问次数

对于常见使用场景，此功能允许对缓存命中率进行分析。

2.6.3.6. XIP 寄存器清单

XIP寄存器起始地址为 `0x14000000`（在SDK中定义为 `XIP_CTRL_BASE`）。

表 154. XIP
寄存器清单

偏移量	名称	说明
0x00	<code>CTRL</code>	缓存控制
0x04	<code>刷新</code>	缓存刷新控制
0x08	<code>状态</code>	缓存状态
0x0c	<code>CTR_HIT</code>	缓存命中计数器
0x10	<code>CTR_ACC</code>	缓存访问计数器
0x14	<code>STREAM_ADDR</code>	FIFO流地址
0x18	<code>STREAM_CTR</code>	FIFO流控制
0x1c	<code>STREAM_FIFO</code>	FIFO流数据

XIP：CTRL 寄存器

偏移：0x00

描述

缓存控制

Table 155. CTRL Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	POWER_DOWN: When 1, the cache memories are powered down. They retain state, but can not be accessed. This reduces static power dissipation. Writing 1 to this bit forces CTRL_EN to 0, i.e. the cache cannot be enabled when powered down. Cache-as-SRAM accesses will produce a bus error response when the cache is powered down.	RW	0x0
2	Reserved.	-	-
1	ERR_BADWRITE: When 1, writes to any alias other than 0x0 (caching, allocating) will produce a bus fault. When 0, these writes are silently ignored. In either case, writes to the 0x0 alias will deallocate on tag match, as usual.	RW	0x1
0	EN: When 1, enable the cache. When the cache is disabled, all XIP accesses will go straight to the flash, without querying the cache. When enabled, cacheable XIP accesses will query the cache, and the flash will not be accessed if the tag matches and the valid bit is set. If the cache is enabled, cache-as-SRAM accesses have no effect on the cache data RAM, and will produce a bus error response.	RW	0x1

XIP: FLUSH Register

Offset: 0x04

Description

Cache Flush control

Table 156. FLUSH Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Write 1 to flush the cache. This clears the tag memory, but the data memory retains its contents. (This means cache-as-SRAM contents is not affected by flush or reset.) Reading will hold the bus (stall the processor) until the flush completes. Alternatively STAT can be polled until completion.	SC	0x0

XIP: STAT Register

Offset: 0x08

Description

Cache Status

Table 157. STAT Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	FIFO_FULL: When 1, indicates the XIP streaming FIFO is completely full. The streaming FIFO is 2 entries deep, so the full and empty flag allow its level to be ascertained.	RO	0x0
1	FIFO_EMPTY: When 1, indicates the XIP streaming FIFO is completely empty.	RO	0x1

表 155. CTRL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	POWER_DOWN : 当该位为1时，缓存内存断电，但状态保持，但无法访问。这降低了静态功耗。 向此位写入1将使 CTRL_EN 强制为0，即断电时缓存无法启用。 缓存断电时，作为SRAM使用的缓存访问将触发总线错误响应。	读写	0x0
2	保留。	-	-
1	ERR_BADWRITE : 当值为1时，对除0x0（缓存、分配）外的任何别名的写入将产生总线错误。当值为0时，此类写入将被静默忽略。 在任何情况下，对0x0别名的写入在标签匹配时均按惯例进行取消分配。	读写	0x1
0	EN : 当值为1时，启用缓存。缓存禁用时，所有 XIP 访问将直接访问闪存，不查询缓存。启用缓存后，可缓存的 XIP 访问将查询缓存；当标签匹配且有效位被设置时，闪存不会被访问。 缓存启用时，作为SRAM的缓存访问不会影响缓存数据RAM，且会触发总线错误响应。	读写	0x1

XIP: FLUSH寄存器

偏移量: 0x04

描述

缓存刷新控制

表 156. FLUSH
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	写入1以执行缓存刷新。此操作会清除标签存储器，但数据存储器内容保持不变。（这意味着作为SRAM使用的缓存内容不会受到刷新或复位的影响。） 读取操作将占用总线（阻塞处理器），直至刷新完成。或者可轮询STAT寄存器，直至刷新完成。	SC	0x0

XIP: STAT寄存器

偏移量: 0x08

描述

缓存状态

表 157. STAT
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	FIFO_FULL : 该位为1时，表示XIP流FIFO已满。 流FIFO深度为2，因此满标志和空标志可用以判断FIFO中的数据量。	只读	0x0
1	FIFO_EMPTY : 该位为1时，表示XIP流FIFO为空。	只读	0x1

Bits	Description	Type	Reset
0	FLUSH_READY: Reads as 0 while a cache flush is in progress, and 1 otherwise. The cache is flushed whenever the XIP block is reset, and also when requested via the FLUSH register.	RO	0x0

XIP: CTR_HIT Register

Offset: 0x0c

Description

Cache Hit counter

Table 158. CTR_HIT Register

Bits	Description	Type	Reset
31:0	A 32 bit saturating counter that increments upon each cache hit, i.e. when an XIP access is serviced directly from cached data. Write any value to clear.	WC	0x00000000

XIP: CTR_ACC Register

Offset: 0x10

Description

Cache Access counter

Table 159. CTR_ACC Register

Bits	Description	Type	Reset
31:0	A 32 bit saturating counter that increments upon each XIP access, whether the cache is hit or not. This includes noncacheable accesses. Write any value to clear.	WC	0x00000000

XIP: STREAM_ADDR Register

Offset: 0x14

Description

FIFO stream address

Table 160.
STREAM_ADDR Register

Bits	Description	Type	Reset
31:2	The address of the next word to be streamed from flash to the streaming FIFO. Increments automatically after each flash access. Write the initial access address here before starting a streaming read.	RW	0x00000000
1:0	Reserved.	-	-

XIP: STREAM_CTR Register

Offset: 0x18

Description

FIFO stream control

Table 161.
STREAM_CTR Register

Bits	Description	Type	Reset
31:22	Reserved.	-	-

位	描述	类型	复位值
0	FLUSH_READY : 缓存刷新进行时该位为0，完成后为1。 每当XIP块重置时，缓存即被刷新，亦可通过FLUSH寄存器请求刷新。	只读	0x0

XIP: CTR_HIT寄存器

偏移: 0x0c

描述

缓存命中计数器

表158. CTR_HIT
寄存器

位	描述	类型	复位值
31:0	32位饱和计数器，每当缓存命中时递增，即XIP访问直接由缓存数据提供服务时。 写入任意值可清零。	WC	0x00000000

XIP: CTR_ACC寄存器

偏移: 0x10

描述

缓存访问计数器

表159. CTR_ACC
寄存器

位	描述	类型	复位值
31:0	32位饱和计数器，每当发生XIP访问时递增，无论缓存是否命中，包含非缓存访问。 写入任意值可清零。	WC	0x00000000

XIP: STREAM_ADDR寄存器

偏移: 0x14

描述

FIFO流地址

表160.
STREAM_ADDR
寄存器

位	描述	类型	复位值
31:2	下一字的地址，将从闪存流式传输至FIFO。 每次闪存访问后自动递增。 在开始流式读取前，需将初始访问地址写入此处。	读写	0x00000000
1:0	保留。	-	-

XIP: STREAM_CTR 寄存器

偏移量: 0x18

描述

FIFO 流控制

表 161.
STREAM_CTR 寄存器

位	描述	类型	复位值
31:22	保留。	-	-

Bits	Description	Type	Reset
21:0	Write a nonzero value to start a streaming read. This will then progress in the background, using flash idle cycles to transfer a linear data block from flash to the streaming FIFO. Decrement automatically (1 at a time) as the stream progresses, and halts on reaching 0. Write 0 to halt an in-progress stream, and discard any in-flight read, so that a new stream can immediately be started (after draining the FIFO and reinitialising STREAM_ADDR)	RW	0x000000

XIP: STREAM_FIFO Register

Offset: 0x1c

Description

FIFO stream data

Table 162.
STREAM_FIFO
Register

Bits	Description	Type	Reset
31:0	Streamed data is buffered here, for retrieval by the system DMA. This FIFO can also be accessed via the XIP_AUX slave, to avoid exposing the DMA to bus stalls caused by other XIP traffic.	RF	0x00000000

2.7. Boot Sequence

Several components of the RP2040 work together to get to a point where the processors are out of reset and able to run the bootrom ([Section 2.8](#)). The bootrom is software that is built into the chip, performing the "processor controlled" part of the boot sequence. We will refer to the steps before the processor is running as the "hardware controlled" boot sequence.

The hardware controlled boot sequence is as follows:

- Power is applied to the chip and the **RUN** pin is high. (If **RUN** is low then the chip will be held in reset.)
- The On-Chip Voltage Regulator ([Section 2.10](#)) waits until the digital core supply (DVDD) is stable
- The Power-On State Machine ([Section 2.13](#)) is started. To summarise the sequence:
 - The Ring Oscillator ([Section 2.17](#)) is started, providing a clock source to the clock generators. `clk_sys` and `clk_ref` are now running at a relatively low frequency (typically 6.5MHz).
 - The reset controller ([Section 2.14](#)), the execute-in-place hardware ([Section 2.6.3](#)), memories ([Section 2.6.2](#) and [Section 2.6.1](#)), Bus Fabric ([Section 2.1](#)), and Processor Subsystem ([Section 2.3](#)) are taken out of reset.
 - Processor core 0 and core 1 begin to execute the bootrom ([Section 2.8](#)).

2.8. Bootrom

The Bootrom size is limited to 16kB. It contains:

- Processor core 0 initial boot sequence.
- Processor core 1 low power wait and launch protocol.
- USB MSC class-compliant bootloader with **UF2** support for downloading code/data to FLASH or RAM.
- USB PICOBLOCK bootloader interface for advanced management.

位	描述	类型	复位值
21:0	<p>写入非零值以启动流式读取。随后该过程将在后台进行，利用闪存空闲周期将线性数据块从闪存传输至流式 FIFO。</p> <p>随着数据流进展自动递减（每次递减 1），并在达到 0 时停止。</p> <p>写入 0 以停止正在进行的流，并丢弃所有在途读取，以便立即启动新流（在排空 FIFO 并重新初始化 STREAM_ADDR 后）</p> <ul style="list-style-type: none"> ◦ 	读写	0x000000

XIP: STREAM_FIFO 寄存器

偏移量: 0x1c

描述

FIFO 流数据

表 162.
STREAM_FIFO
寄存器

位	描述	类型	复位值
31:0	流式数据在此缓冲，供系统 DMA 读取。 该 FIFO 也可通过 XIP_AUX 从属访问，避免 DMA 因其他 XIP 访问导致的总线阻塞。	RF	0x00000000

2.7. 启动顺序

RP2040 的多个组件协同工作，使处理器脱离复位状态并能执行 bootrom（第 2.8 节）。bootrom 是内置芯片的软件，负责执行“处理器控制”的启动序列部分。我们将处理器启动前的步骤称为“硬件控制”启动序列。

硬件控制启动序列如下：

- 电源施加至芯片，RUN 引脚为高电平。（如 RUN 为低电平，芯片将保持复位状态。）
- 片上电压调节器（第 2.10 节）等待数字核心电源（DVDD）稳定。
- 上电状态机（第 2.13 节）启动。序列总结如下：
 - 环形振荡器（第 2.17 节）启动，为时钟发生器提供时钟源。clk_sys 和 clk_ref 现以相对较低频率（通常为 6.5MHz）运行。
 - 复位控制器（第 2.14 节）、原地执行硬件（第 2.6.3 节）、存储器（第 2.6.2 节和第 2.6.1 节）、总线结构（第 2.1 节）及处理器子系统（第 2.3 节）已退出复位状态。
 - 处理器核心 0 和核心 1 开始执行 Bootrom（第 2.8 节）。

2.8. 启动只读存储器 (Bootrom)

Bootrom 的大小限制为 16kB。其内容包括：

- 处理器核心 0 的初始启动序列。
- 处理器核心 1 的低功耗等待与启动协议。
- 支持 UF2 的 USB MSC 类兼容引导加载程序，用于向 FLASH 或 RAM 下载代码和数据。
- 用于高级管理的 USB PICOBOOT 引导加载程序接口。

- Routines for programming and manipulating the external flash.
- Fast floating point library.
- Fast bit counting / manipulation functions.
- Fast memory fill / copy functions.

Bootrom Source Code

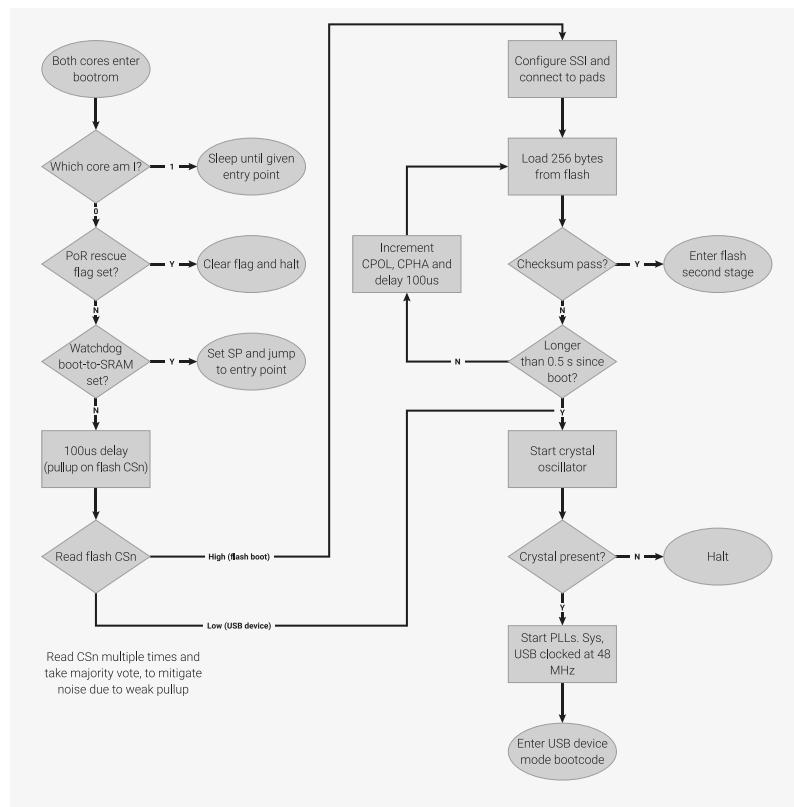
The full source for the RP2040 bootrom can be found at <https://github.com/raspberrypi/pico-bootrom>.

This includes versions 1, 2 and 3 of the bootrom, which correspond to the B0, B1 and B2 silicon revisions, respectively.

2.8.1. Processor Controlled Boot Sequence

A flow diagram of the boot sequence is given in [Figure 15](#).

Figure 15. RP2040 Boot Sequence



After the hardware controlled boot sequence described in [Section 2.7](#), the processor controlled boot sequence starts:

- Reset to both processors released: both enter ROM at same location
- Processors check SIO.CPUID
 - Processor 1 goes to sleep (WFE with SCR.SLEEPDEEP enabled) and remains asleep until woken by user code, via the mailbox
 - Processor 0 continues executing from ROM
- If power up event was from Rescue DP, clear this flag and **halt immediately**
 - The debug host (which initiated the rescue) will provide further instruction.
- If watchdog scratch registers set to indicate pre-loaded code exists in SRAM, jump to that code

- 用于编程和操作外部闪存的例程。
 - 快速浮点库。
 - 快速位计数与操作函数。
 - 快速内存填充与复制函数。

Bootrom源代码

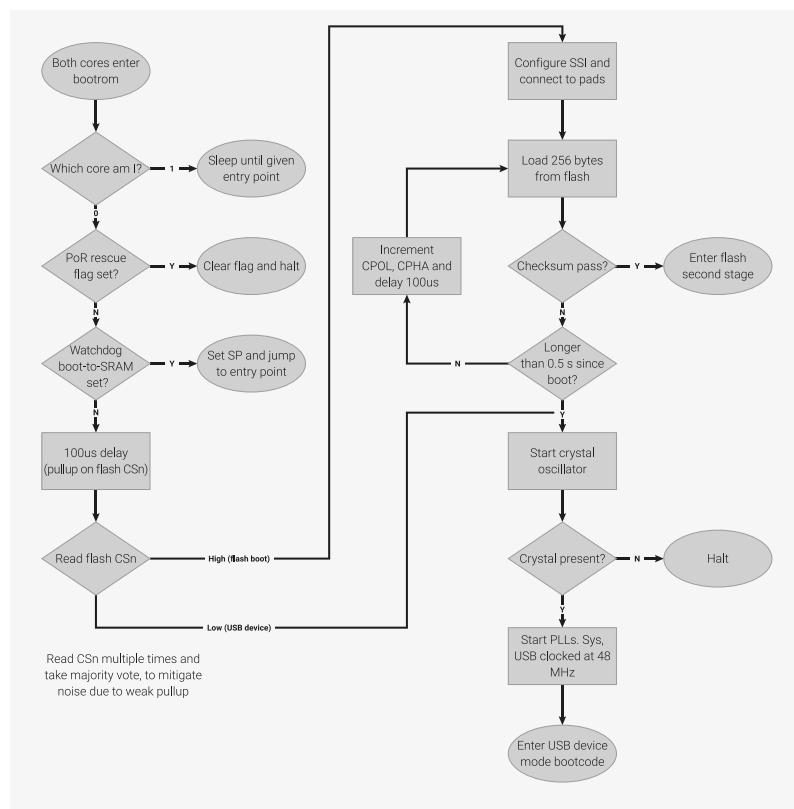
RP2040 Bootrom的完整源代码可于<https://github.com/raspberrypi/pico-bootrom>获取。

其中包含bootrom的第1版、第2版和第3版，分别对应B0、B1和B2硅片
修订版本。

2.8.1. 处理器控制的启动顺序

启动序列的流程图参见图15。

图15. RP2040
启动序列



在第2.7节所述的硬件控制启动序列之后，处理器控制启动序列开始执行：

- 两个处理器复位释放：均从相同地址进入ROM
 - 处理器检查 SIO.CPUD
 - 处理器1进入休眠状态（启用SCR.SLEEPDEEP的WFE），并持续休眠，直至通过邮箱由用户代码唤醒
 - 处理器0继续从ROM执行
 - 若上电事件源自救援DP，清除此标志并**立即停止**
 - 发起救援的调试主机将提供后续指令。
 - 若看门狗擦写寄存器指示SRAM内存在预加载代码，则跳转至该代码

- Check if SPI CS pin is tied low ("bootrom button"), and skip flash boot if so.
- Set up IO muxing, pad controls on QSPI pins, and initialise Synopsys SSI for standard SPI mode
- Issue XIP exit sequence, in case flash is still in an XIP mode and has not been power-cycled
- Copy 256 bytes from SPI to internal SRAM (SRAM5) and check for valid CRC32 checksum
- If checksum passes, assume what we have loaded is a valid flash second stage
- Start executing the loaded code from SRAM (SRAM5)
- If no valid image found in SPI after 0.5 seconds of attempting to boot, drop to USB device boot
- USB device boot: appear as a USB Mass Storage Device
 - Can program the SPI flash, or load directly into SRAM and run, by dragging and dropping an image in UF2 format.
 - Also supports an extended PICOBLOCK interface

2.8.1.1. Watchdog Boot

Watchdog boot allows users to install their own boot handler, and divert control away from the main boot sequence on non-POR/BOR resets. It also simplifies running code over the JTAG test interface. It recognises the following values written to the watchdog's upper scratch registers:

- Scratch 4: magic number `0xb007c0d3`
- Scratch 5: Entry point XORed with magic `-0xb007c0d3 (0x4ff83f2d)`
- Scratch 6: Stack pointer
- Scratch 7: Entry point

If either of the magic numbers mismatch, watchdog boot does not take place. If the numbers match, the Bootrom zeroes scratch 4 before transferring control, so that the behaviour does not persist over subsequent reboots.

2.8.1.2. Flash Boot Sequence

One of the main challenges of a warm flash boot is forcing the external flash from XIP mode to a mode where it will accept standard SPI commands. There is no standard method to discontinue XIP on an unknown flash. The Bootrom provides a best-effort sequence with broad compatibility, which is as follows:

- `CSn=1, IO[3:0]=4'b0000` (via pull-downs to avoid contention), issue ×32 clocks
- `CSn=0, IO[3:0]=4'b1111` (via pull-ups to avoid contention), issue ×32 clocks
- `CSn=1`
- `CSn=0, MOSI=1'b1` (driven low-Z, all other IOs Hi-Z), issue ×16 clocks

This is designed to miss the XIP continuation codes on Cypress, Micron and Winbond parts. If the device is already in SPI mode, it interprets this sequence as two `FFh NOP` instructions, which should be ignored.

As this is best effort only, there may be some devices which obstinately remain in XIP mode. There are then two options:

- Use a less efficient XIP mode where each transfer has an SPI instruction prefix, so the flash device remains communicative in SPI mode.
- Boot code installs a compatible XIP exit sequence in SRAM, and configures the watchdog such that a warm boot will jump straight into this sequence, foregoing our canned sequence.

After issuing the XIP exit sequence, the Bootrom attempts to read in the second stage from flash using standard `03h` serial read commands, which are near-universally supported. Since the Bootrom is immutable, it aims for compatibility

- 检查SPI CS引脚是否被拉低（“bootrom按钮”），若是则跳过闪存启动。
- 设置QSPI引脚的IO复用及管脚控制，初始化Synopsys SSI以使用标准SPI模式
- 发出XIP退出序列，以防闪存仍处于XIP模式且未断电
- 从 SPI 复制 256 字节至内部 SRAM (SRAM5)，并验证 CRC32 校验和的有效性
- 若校验和通过，则视所加载内容为有效的闪存第二阶段程序
- 开始从 SRAM (SRAM5) 执行已加载代码
- 若在尝试启动后 0.5 秒内未在 SPI 中找到有效映像，则切换至 USB 设备启动
- USB 设备启动：作为 USB 大容量存储设备出现
 - 可通过拖放 UF2 格式映像文件编程 SPI 闪存，或直接加载至 SRAM 并运行
 - 亦支持扩展的 PICOBLOCK 接口

2.8.1.1. 看门狗启动

看门狗启动允许用户安装自定义启动处理程序，并在非上电复位 (POR) 或欠压复位 (BOR) 时，将控制权从主启动序列分离同时简化了通过 JTAG 测试接口运行代码的操作系统识别写入看门狗高位 scratch 寄存器的以下数值：

- Scratch 4：魔术数字 `0xb007c0d3`
- Scratch 5：入口点与魔数异或 `-0xb007c0d3 (0x4ff83f2d)`
- Scratch 6：栈指针
- Scratch 7：入口点

如果任一魔数不匹配，则不会执行看门狗启动。若数字匹配，Bootrom 会在转移控制权前清零 scratch 4，以防该行为延续至后续重启。

2.8.1.2. 闪存启动序列

热启动闪存的主要挑战之一是强制外部闪存从 XIP 模式切换至可接受标准 SPI 命令的模式。针对未知闪存，无标准方法可中断 XIP。Bootrom 提供了兼容性广泛的尽力而为序列，具体如下：

- `CSn=1, IO[3:0]=4'b0000` (通过下拉电阻避免争用)，发出 ×32 时钟
- `CSn=0, IO[3:0]=4'b1111` (通过上拉避免争用)，发出 ×32 个时钟
- `CSn=1`
- `CSn=0, MOSI=1'b1` (低阻驱动，所有其他IO为高阻)，发出 ×16 个时钟

此设计旨在避开Cypress、Micron及Winbond芯片上的XIP续传代码。如果设备已处于SPI模式，则将此序列解释为两个`FFh NOP`指令，应予以忽略。

鉴于此仅为尽力而为，可能存在某些设备固执地维持在XIP模式。因此有以下两种选择：

- 使用效率较低的XIP模式，其中每次传输均带有SPI指令前缀，使闪存设备保持SPI模式下的通信。
- 引导代码在SRAM中安装兼容的XIP退出序列，并配置看门狗，使得热启动时能直接跳转至该序列，放弃我们的预设序列。

发出XIP退出序列后，Bootrom尝试使用标准的 `03h`串行读取命令从闪存读取第二阶段代码，该命令几乎得到了普遍支持。由于Bootrom不可更改，其目标是实现兼容性

rather than performance.

2.8.1.3. Flash Second Stage

The flash second stage must configure the SSI and the external flash for the best possible execute-in-place performance. This includes interface width, SCK frequency, SPI instruction prefix and an XIP continuation code for address-data only modes. Generally some operation can be performed on the external flash so that it does not require an instruction prefix on each access, and will simply respond to addresses with data.

Until the SSI is correctly configured for the attached flash device, it is not possible to access flash via the XIP address window. Additionally, the Synopsys SSI can not be reconfigured at all without first disabling it. Therefore the second stage must be copied from flash to SRAM by the bootrom, and executed in SRAM.

Alternatively, the second stage can simply shadow an image from external flash into SRAM, and not configure execute-in-place.

This is the **only** job of the second stage. All other chip setup (e.g. PLLs, Voltage Regulator) can be performed by platform initialisation code executed over the XIP interface, once the second stage has run.

2.8.1.3.1. Checksum

The last four bytes of the image loaded from flash (which we hope is a valid flash second stage) are a CRC32 checksum of the first 252 bytes. The parameters of the checksum are:

- Polynomial: `0x04c11db7`
- Input reflection: no
- Output reflection: no
- Initial value: `0xffffffff`
- Final XOR: `0x00000000`
- Checksum value appears as little-endian integer at end of image

The Bootrom makes 128 attempts of approximately 4ms each for a total of approximately 0.5 seconds before giving up and dropping into USB code to load and checksum the second stage with varying SPI parameters. If it sees a checksum pass it will immediately jump into the 252-byte payload which contains the flash second stage.

2.8.2. Launching Code On Processor Core 1

As described in the introduction to [Section 2.8.1](#), after reset, processor core 1 "sleeps (WFE with SCR.SLEEPDEEP enabled) and remains asleep until woken by user code, via the mailbox".

If you are using the SDK then you can simply use the `multicore_launch_core1` function to launch code on processor core 1. However this section describes the procedure to launch code on processor core 1 yourself.

The procedure to start running on processor core 1 involves both cores moving in lockstep through a state machine coordinated by passing messages over the inter-processor FIFOs. This state machine is designed to be robust enough to cope with a recently reset processor core 1 which may be anywhere in its boot code, up to and including going to sleep. As result, the procedure may be performed at any point after processor core 1 has been reset (either by system reset, or explicitly resetting just processor core 1).

The following C code is the simplest way to describe the procedure:

```
// values to be sent in order over the FIFO from core 0 to core 1
//
// vector_table is value for VTOR register
```

而非性能优化。

2.8.1.3. 闪存第二阶段

闪存第二阶段必须为SSI和外部闪存配置，以实现最佳的执行就地（execute-in-place）性能。这包括接口宽度、SCK频率、SPI指令前缀及用于地址-数据模式的XIP续传代码。通常可以对外部闪存执行特定操作，使其无需在每次访问时使用指令前缀，仅通过地址即可响应数据。

在SSI未正确配置对应闪存设备之前，无法通过XIP地址窗口访问闪存。此外，Synopsys SSI在未先禁用前无法进行重新配置。因此，第二阶段代码必须由Bootrom从闪存复制至SRAM，并在SRAM中执行。

或者，第二阶段可以简单地将外部闪存中的镜像映射（shadow）到SRAM中，而不配置执行原地（execute-in-place）。

这是第二阶段的唯一任务。所有其他芯片设置（如PLL、电压调节器）可由第二阶段执行完成后，通过XIP接口运行的平台初始化代码进行。

2.8.1.3.1. 校验和

从闪存加载的镜像的最后四个字节（预期为有效的闪存第二阶段）是前252个字节的CRC32校验和。校验和的参数如下：

- 多项式：`0x04c11db7`
- 输入反射：否
- 输出反射：否
- 初始值：`0xffffffff`
- 最终异或：`0x00000000`
- 校验和值以小端整数形式出现在镜像末尾。

Bootrom会进行约128次尝试，每次约4毫秒，总计约0.5秒，尝试失败后放弃，转而执行USB代码以不同SPI参数加载并校验第二阶段。如果检测到校验和通过，它将立即跳转到包含闪存第二阶段的252字节负载部分。

2.8.2. 在处理器核1上启动代码

如第2.8.1节介绍所述，复位后，处理器核心1将“休眠（启用SCR.SLEEPDEEP的WFE）”，并保持休眠状态，直到由用户代码通过邮箱唤醒。

如果您使用SDK，则可以直接调用`multicore_launch_core1`函数在处理器核心1上启动代码。
然而，本节描述了自行启动处理器核心1上代码的具体流程。

启动处理器核心1的过程涉及两个核心通过互处理器FIFO传递消息，以锁步方式驱动状态机运行。该状态机设计足够健壮，能够应对刚复位、并可能处于启动代码任意阶段（包括进入休眠状态）的处理器核心1。因此，该过程可在处理器核心1复位后任何时刻执行，无论是系统复位还是仅显式复位处理器核心1。

以下C代码为描述该过程的最简方式：

```
// 通过FIFO从核心0到核心1按顺序发送的值
//
// vector_table为VTOR寄存器的值
```

```

// sp is initial stack pointer (SP)
// entry is the initial program counter (PC) (don't forget to set the thumb bit!)
const uint32_t cmd_sequence[] =
    {0, 0, 1, (uintptr_t) vector_table, (uintptr_t) sp, (uintptr_t) entry};

uint seq = 0;
do {
    uint cmd = cmd_sequence[seq];
    // always drain the READ FIFO (from core 1) before sending a 0
    if (!cmd) {
        // discard data from read FIFO until empty
        multicore_fifo_drain();
        // execute a SEV as core 1 may be waiting for FIFO space
        __sev();
    }
    // write 32 bit value to write FIFO
    multicore_fifo_push_blocking(cmd);
    // read 32 bit value from read FIFO once available
    uint32_t response = multicore_fifo_pop_blocking();
    // move to next state on correct response (echo-d value) otherwise start over
    seq = cmd == response ? seq + 1 : 0;
} while (seq < count_of(cmd_sequence));

```

2.8.3. Bootrom Contents

Some of the bootrom is dedicated to the implementation of the boot sequence and USB boot interfaces. There is also code in the bootrom useful to user programs. [Table 163](#) shows the fixed memory layout of the first handful of words in the Bootrom which are instrumental in locating other content within the bootrom.

Table 163. Bootrom contents at fixed (well known) addresses

Address	Contents	Description
0x00000000	32-bit pointer	Initial boot stack pointer
0x00000004	32-bit pointer	Pointer to boot reset handler function
0x00000008	32-bit pointer	Pointer to boot NMI handler function
0x0000000c	32-bit pointer	Pointer to boot Hard fault handler function
0x00000010	'M', 'u', 0x01	Magic
0x00000013	byte	Bootrom version
0x00000014	16-bit pointer	Pointer to a public function lookup table (<code>rom_func_table</code>)
0x00000016	16-bit pointer	Pointer to a public data lookup table (<code>rom_data_table</code>)
0x00000018	16-bit pointer	Pointer to a helper function (<code>rom_table_lookup()</code>)

2.8.3.1. Bootrom Functions

The Bootrom contains a number of public functions that provide useful RP2040 functionality that might be needed in the absence of any other code on the device, as well as highly optimized versions of certain key functionality that would otherwise have to take up space in most user binaries.

These functions are normally made available to the user by the SDK, however a lower level method is provided to locate them (their locations may change with each Bootrom release) and call them directly.

Assuming the three bytes starting at address `0x00000010` are ('M', 'u', 0x01) then the three halfwords starting at offset `0x00000014` are valid.

```

// sp为初始堆栈指针 (SP)
// entry为初始程序计数器 (PC) (请勿忘记设置Thumb位)
const uint32_t cmd_sequence[] =
{0, 0, 1, (uintptr_t)vector_table, (uintptr_t)sp, (uintptr_t)entry};

uint seq = 0;
do {
    uint cmd = cmd_sequence[seq];
    // 发送0之前应始终清空READ FIFO (来自核心1)
    if (!cmd) {
        // 丢弃READ FIFO中数据直至清空
        multicore_fifo_drain();
        // 执行SEV，核心1可能在等待FIFO空间
        __sev();
    }
    // 向写FIFO写入32位值
    multicore_fifo_push_blocking(cmd);
    // 从读FIFO读取32位值，一旦可用 uint32_t response = multicore_fifo_pop_blocking();
    // 在正确响应 (echo-d值) 时切换到下一个状态，否则重新开始
    seq = cmd == response ? seq + 1 : 0;
} while (seq < count_of(cmd_sequence));

```

2.8.3. 启动只读存储器内容

Bootrom的部分内容专门用于实现启动序列及USB启动接口。Bootrom中也包含对用户程序有用的代码。表163显示了Bootrom前几个固定存储单元的内存布局，这些内容对于定位Bootrom内的其他部分至关重要。

表163。Bootrom
内容在固定（众所
周知）地址处

地址	内容	描述
0x00000000	32位指针	初始启动堆栈指针
0x00000004	32位指针	指向启动复位处理函数的指针
0x00000008	32位指针	指向启动NMI处理函数的指针
0x0000000c	32位指针	指向启动硬故障处理函数的指针
0x00000010	'M', 'u', 0x01	魔术
0x00000013	字节	Bootrom 版本
0x00000014	16 位指针	指向公共函数查找表 (<code>rom_func_table</code>) 的指针
0x00000016	16 位指针	指向公共数据查找表 (<code>rom_data_table</code>) 的指针
0x00000018	16 位指针	指向辅助函数 (<code>rom_table_lookup()</code>) 的指针

2.8.3.1. Bootrom 函数

Bootrom 包含若干公共函数，这些函数提供了在设备缺乏其他代码时可能需要的有用 RP2040 功能，以及某些关键功能的高度优化版本，否则这些功能通常会占用大多数用户二进制文件的空间。

这些函数通常由 SDK 向用户提供，然而也提供了一种较低级的方法以定位它们（位置可能随每个 Bootrom 版本变更）并直接调用。

假设地址 `0x00000010`起始的三个字节为 ('M', 'u', 0x01)，则偏移量 `0x00000014`处的三个半字 (halfwords) 有效。

These three values can be used to dynamically locate other functions or data within the Bootrom. The version byte at offset `0x00000013` is informational and should not be used to infer the exact location of any functions.

The following code from the SDK shows how the three 16-bit pointers are used to lookup other functions or data.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_bootrom/bootrom.c Lines 12 - 19

```

12 void *rom_func_lookup(uint32_t code) {
13     return rom_func_lookup_inline(code);
14 }
15
16 void *rom_data_lookup(uint32_t code) {
17     return rom_data_lookup_inline(code);
18 }
```

The `code` parameter correspond to the *CODE* values in the tables below, and is calculated as follows:

```

uint32_t rom_table_code(char c1, char c2) {
    return (c2 << 8) | c1;
}
```

2.8.3.1.1. Fast Bit Counting / Manipulation Functions

These are optimized versions of common bit counting / manipulation functions.

In general you do not need to call these methods directly as the SDK *pico_bit_ops* library replaces the corresponding standard compiler library functions by default so that the standard functions such as `__builtin_popcount` or `__clzdi2` uses the corresponding Bootrom implementations automatically (see [pico_bit_ops](#) for more details).

These functions have changed in speed slightly between version 1 (V1) of the bootrom and version 2 (V2).

Table 164. Fast Bit Counting / Manipulation Functions.

CODE	Cycles Avg V1	Cycles Avg V2/V3	Description
'P','3'	18	20	<code>uint32_t __popcount32(uint32_t value)</code>
			Return a count of the number of 1 bits in <code>value</code> .
'R','3'	21	22	<code>uint32_t __reverse32(uint32_t value)</code>
			Return the bits of <code>value</code> in the reverse order.
'L','3'	13	9.6	<code>uint32_t __clz32(uint32_t value)</code>
			Return the number of consecutive high order 0 bits of <code>value</code> . If <code>value</code> is zero, returns 32.
'T','3'	12	11	<code>uint32_t __ctz32(uint32_t value)</code>
			Return the number of consecutive low order 0 bits of <code>value</code> . If <code>value</code> is zero, returns 32.

2.8.3.1.2. Fast Bulk Memory Fill / Copy Functions

These are highly optimized bulk memory fill and copy functions commonly provided by most language runtimes.

In general you do not need to call these methods directly as the SDK *pico_mem_ops* library replaces the corresponding standard ARM EABI functions by default so that the standard C library functions e.g. `memcpy` or `memset` use the Bootrom implementations automatically (see [pico_mem_ops](#) for more details).

这三个值可用于动态定位 Bootrom 中的其他函数或数据。偏移量 `0x00000013` 处的版本字节仅供参考，不应作为推断任何函数精确位置的依据。

以下 SDK 代码示例展示了如何使用这三个 16 位指针查找其他函数或数据。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_bootrom/bootrom.c 第 12 至 19 行

```

12 void *rom_func_lookup(uint32_t code) {
13     return rom_func_lookup_inline(code);
14 }
15
16 void *rom_data_lookup(uint32_t code) {
17     return rom_data_lookup_inline(code);
18 }
```

参数 `code` 对应下表中的 `CODE` 值，计算方式如下：

```

uint32_t rom_table_code(char c1, char c2) {
    return (c2 << 8) | c1;
}
```

2.8.3.1.1. 快速位计数／操作函数

这些是常用位计数／操作函数的优化版本。

通常情况下，您无需直接调用这些方法，因为 SDK 中的 `pico_bit_ops` 库默认替换了相应的标准编译器库函数，标准函数如 `_builtin_popcount` 或 `_clzdi2` 会自动调用对应的 Bootrom 实现（详情请参见 `pico_bit_ops`）。

这些函数在 Bootrom 版本 1 (V1) 与版本 2 (V2) 之间的执行速度略有差异。

表164。快速位计数／操作函数。

代码	平均周期数 V1	平均周期数 V2/V3	描述
'P','3'	18	20	<code>uint32_t _popcount32(uint32_t value)</code>
			返回 <code>value</code> 中值为 1 的位数。
'R','3'	21	22	<code>uint32_t _reverse32(uint32_t value)</code>
			返回 <code>value</code> 的位逆序。
'L','3'	13	9.6	<code>uint32_t _clz32(uint32_t value)</code>
			返回 <code>value</code> 中连续的高位 0 的数量。如果 <code>value</code> 为零，则返回 32。
'T','3'	12	11	<code>uint32_t _ctz32(uint32_t value)</code>
			返回 <code>value</code> 中连续的低位 0 的数量。如果 <code>value</code> 为零，则返回 32。

2.8.3.1.2 快速批量内存填充／复制函数

这些函数是大多数语言运行时通常提供的高度优化的批量内存填充和复制函数。

通常情况下，您无需直接调用这些方法，因为 SDK 中的 `pico_mem_ops` 库默认替换了相应的标准 ARM EABI 函数，使得标准 C 库函数如 `memcpy` 或 `memset` 自动使用 Bootrom 实现（详见 `pico_mem_ops`）。

Table 165. Optimized
Bulk Memory Fill /
Copy Functions

CODE	Description
'M','S'	<code>uint8_t *_memset(uint8_t *ptr, uint8_t c, uint32_t n)</code> Sets <code>n</code> bytes start at <code>ptr</code> to the value <code>c</code> and returns <code>ptr</code> .
'S','4'	<code>uint32_t *_memset4(uint32_t *ptr, uint8_t c, uint32_t n)</code> Sets <code>n</code> bytes start at <code>ptr</code> to the value <code>c</code> and returns <code>ptr</code> . Note this is a slightly more efficient variant of <code>_memset</code> that may only be used if <code>ptr</code> is word aligned.
'M','C'	<code>uint8_t *_memcpy(uint8_t *dest, uint8_t *src, uint32_t n)</code> Copies <code>n</code> bytes starting at <code>src</code> to <code>dest</code> and returns <code>dest</code> . The results are undefined if the regions overlap.
'C','4'	<code>uint8_t *_memcpy4(uint32_t *dest, uint32_t *src, uint32_t n)</code> Copies <code>n</code> bytes starting at <code>src</code> to <code>dest</code> and returns <code>dest</code> . The results are undefined if the regions overlap. Note this is a slightly more efficient variant of <code>_memcpy</code> that may only be used if <code>dest</code> and <code>src</code> are word aligned.

2.8.3.1.3. Flash Access Functions

These are low level flash helper functions.

Table 166. Flash
Access Functions

CODE	Description
'I','F'	<code>void _connect_internal_flash(void)</code> Restore all QSPI pad controls to their default state, and connect the SSI to the QSPI pads
'E','X'	<code>void _flash_exit_xip(void)</code> First set up the SSI for serial-mode operations, then issue the fixed XIP exit sequence described in Section 2.8.1.2. Note that the bootrom code uses the IO forcing logic to drive the CS pin, which must be cleared before returning the SSI to XIP mode (e.g. by a call to <code>_flash_flush_cache</code>). This function configures the SSI with a fixed SCK clock divisor of /6.
'R','E'	<code>void _flash_range_erase(uint32_t addr, size_t count, uint32_t block_size, uint8_t block_cmd)</code> Erase a <code>count</code> bytes, starting at <code>addr</code> (offset from start of flash). Optionally, pass a block erase command e.g. <code>D8h block erase</code> , and the size of the block erased by this command – this function will use the larger block erase where possible, for much higher erase speed. <code>addr</code> must be aligned to a 4096-byte sector, and <code>count</code> must be a multiple of 4096 bytes.
'R','P'	<code>void flash_range_program(uint32_t addr, const uint8_t *data, size_t count)</code> Program <code>data</code> to a range of flash addresses starting at <code>addr</code> (offset from the start of flash) and <code>count</code> bytes in size. <code>addr</code> must be aligned to a 256-byte boundary, and <code>count</code> must be a multiple of 256.
'F','C'	<code>void _flash_flush_cache(void)</code> Flush and enable the XIP cache. Also clears the IO forcing on QSPI CSn, so that the SSI can drive the flash chip select as normal.
'C','X'	<code>void _flash_enter_cmd_xip(void)</code> Configure the SSI to generate a standard <code>03h</code> serial read command, with 24 address bits, upon each XIP access. This is a very slow XIP configuration, but is very widely supported. The debugger calls this function after performing a flash erase/programming operation, so that the freshly-programmed code and data is visible to the debug host, without having to know exactly what kind of flash device is connected.

A typical call sequence for erasing a flash sector from user code would be:

表 165。优化的批量内存填充 /
复制函数

代码	描述
'M','S'	<code>uint8_t *_memset(uint8_t *ptr, uint8_t c, uint32_t n)</code> 将从 <code>ptr</code> 开始的 <code>n</code> 个字节设置为值 <code>c</code> ，返回 <code>ptr</code> 。
'S','4'	<code>uint32_t *_memset4(uint32_t *ptr, uint8_t c, uint32_t n)</code> 将从 <code>ptr</code> 开始的 <code>n</code> 个字节设置为值 <code>c</code> ，返回 <code>ptr</code> 。注意，此函数为 <code>_memset</code> 的稍高效变体，仅当 <code>ptr</code> 按字对齐时方可使用。
'M','C'	<code>uint8_t *_memcpy(uint8_t *dest, uint8_t *src, uint32_t n)</code> 将从 <code>src</code> 开始的 <code>n</code> 个字节复制到 <code>dest</code> ，返回 <code>dest</code> 。若内存区域重叠，结果未定义。
'C','4'	<code>uint8_t *_memcpy44(uint32_t *dest, uint32_t *src, uint32_t n)</code> 将从 <code>src</code> 开始的 <code>n</code> 个字节复制到 <code>dest</code> ，返回 <code>dest</code> 。若内存区域重叠，结果未定义。 注意，此函数为 <code>_memcpy</code> 的稍高效变体，仅当 <code>dest</code> 和 <code>src</code> 均按字对齐时方可使用。

2.8.3.1.3. 闪存访问函数

这些是底层的闪存辅助函数。

表 166. 闪存
访问函数

代码	描述
'I','F'	<code>void _connect_internal_flash(void)</code> 将所有 QSPI 引脚控制恢复至默认状态，并将 SSI 连接至 QSPI 引脚。
'E','X'	<code>void _flash_exit_xip(void)</code> 首先将 SSI 设置为串行模式操作，然后执行第 2.8.1.2 节中描述的固定 XIP 退出序列。请注意，bootrom 代码采用 IO 强制逻辑驱动 CS 引脚，返回 SSI 至 XIP 模式前必须清除此状态（例如通过调用 <code>_flash_flush_cache</code> 函数）。此函数将 SSI 配置为固定的 SCK 时钟分频器，分频值为 /6。
'R','E'	<code>void _flash_range_erase(uint32_t addr, size_t count, uint32_t block_size, uint8_t block_cmd)</code> 从 <code>addr</code> （相对于闪存起始的偏移）开始，擦除 <code>count</code> 字节。可选地，传入块擦除命令，例如 D8h 块擦除，以及该命令擦除的块大小——本函数将在可能的情况下使用更大块擦除，以极大提升擦除速度。 <code>addr</code> 必须按 4096 字节扇区对齐，且 <code>count</code> 必须为 4096 字节的整数倍。
'R','P'	<code>void flash_range_program(uint32_t addr, const uint8_t *data, size_t count)</code> 将 <code>data</code> 编程至以 <code>addr</code> （相对于闪存起始的偏移）为起始地址、大小为 <code>count</code> 字节的闪存区域。 <code>addr</code> 必须按 256 字节边界对齐，且 <code>count</code> 须为 256 的整数倍。
'F','C'	<code>void _flash_flush_cache(void)</code> 刷新并启用 XIP 缓存。同时清除 QSPI CSn 上的 IO 强制，使 SSI 可正常驱动闪存芯片选择信号。
'C','X'	<code>void _flash_enter_cmd_xip(void)</code> 配置 SSI 以生成标准的 03h 串行读取命令，包含 24 位地址，于每次 XIP 访问时触发。此为速度极慢的 XIP 配置，但具有广泛的兼容性。调试器在执行闪存擦除/编程操作后调用此函数，以使新编程的代码和数据对调试主机可见，无需准确了解所连接的闪存设备类型。

从用户代码擦除闪存扇区的典型调用顺序如下：

- `_connect_internal_flash`
- `_flash_exit_xip`
- `_flash_range_erase(addr, 1 << 12, 1 << 16, 0xd8)`
- `_flash_flush_cache`
- Either a call to `_flash_enter_cmd_xip` or call into a flash second stage that was previously copied out into SRAM

Note that, in between the first and last calls in this sequence, the SSI is **not** in a state where it can handle XIP accesses, so the code that calls the intervening functions must be located in SRAM. The SDK `hardware_flash` library hides these details.

2.8.3.1.4. Debugging Support Functions

These two methods simplify the task of calling code on the device and then returning control to the debugger.

Table 167. Debugging Support Functions

CODE	Description
'D', 'T'	<p><code>_debug_trampoline</code></p> <p>Simple debugger trampoline for break-on-return.</p> <p>This method helps the debugger call ROM routines without setting hardware breakpoints. The function address is passed in <code>r7</code> and args are passed through <code>r0 ... r3</code> as per ABI.</p> <p>This method does not return but executes a <code>BKPT #0</code> at the end.</p>
'D', 'E'	<p><code>_debug_trampoline_end</code></p> <p>This is the address of the final <code>BKPT #0</code> instruction of <code>debug_trampoline</code>. This can be compared with the program counter to detect completion of the <code>debug_trampoline</code> call.</p>

2.8.3.1.5. Miscellaneous Functions

These remaining functions don't fit in other categories and are exposed in the SDK via the `pico_bootrom` library (see [pico_bootrom](#)).

Table 168.
Miscellaneous Functions

CODE	Description
'U', 'B'	<p><code>void _reset_to_usb_boot(uint32_t gpio_activity_pin_mask, uint32_t disable_interface_mask)</code></p> <p>Resets the RP2040 and uses the watchdog facility to re-start in BOOTSEL mode:</p> <ul style="list-style-type: none"> • <code>gpio_activity_pin_mask</code> is provided to enable an "activity light" via GPIO attached LED for the USB Mass Storage Device: <ul style="list-style-type: none"> ◦ <code>0</code> No pins are used as per a cold boot. ◦ Otherwise a single bit set indicating which GPIO pin should be set to output and raised whenever there is mass storage activity from the host. • <code>disable_interface_mask</code> may be used to control the exposed USB interfaces: <ul style="list-style-type: none"> ◦ <code>0</code> To enable both interfaces (as per a cold boot) ◦ <code>1</code> To disable the USB Mass Storage Interface (see Section 2.8.4) ◦ <code>2</code> To disable the USB PICOBLOCK Interface (see Section 2.8.5)

- `_connect_internal_flash`
- `_flash_exit_xip`
- `_flash_range_erase(addr, 1 << 12, 1 << 16, 0xd8)`
- `_flash_flush_cache`
- 调用`_flash_enter_cmd_xip`, 或调用先前复制到SRAM的闪存二级启动程序。

请注意，在该序列的首次与末次调用间，SSI处于无法处理XIP访问的状态，因此调用中间函数的代码必须驻留于SRAM中。SDK中的[hardware_flash](#)库封装了相关细节。

2.8.3.1.4. 调试支持功能

这两种方法简化了在设备上调用代码并将控制权返回调试器的流程。

表167。调试支持功能

代码	描述
'D', 'T'	<p><code>_debug_trampoline</code></p> <p>用于返回时设置断点的简易调试跳板。</p> <p>此方法协助调试器在不设置硬件断点的情况下调用ROM例程。函数地址通过 <code>r7</code> 传递，参数通过 <code>r0</code> 至 <code>r3</code> 按 ABI 规定传递。</p> <p>此方法不返回，结束时执行 <code>BKPT #0</code> 指令。</p>
'D', 'E'	<p><code>_debug_trampoline_end</code></p> <p>此地址为 <code>debug_trampoline</code> 中最后一条 <code>BKPT #0</code> 指令的位置。可通过与程序计数器比较，检测 <code>debug_trampoline</code> 调用的完成。</p>

2.8.3.1.5. 杂项功能

这些剩余功能未归类于其他类别，且通过[pico_bootrom](#)库在SDK中提供（详见[pico_bootrom](#)）。

表168。
杂项
函数

代码	描述
'U', 'B'	<p><code>void _reset_to_usb_boot(uint32_t gpio_activity_pin_mask, uint32_t disable_interface_mask)</code></p> <p>重置RP2040并利用看门狗功能重新启动至BOOTSEL模式：</p> <ul style="list-style-type: none"> • <code>gpio_activity_pin_mask</code> 用于通过连接至GPIO的LED启用USB大容量存储设备的“活动指示灯”： <ul style="list-style-type: none"> ◦ 0 不使用任何引脚，等同冷启动。 ◦ 否则为单个位掩码，指示应将哪个GPIO引脚设置为输出，并在主机进行大容量存储活动时将其置高。 • <code>disable_interface_mask</code> 用于控制暴露的USB接口： <ul style="list-style-type: none"> ◦ 0 启用两个接口（等同冷启动） ◦ 1 禁用USB大容量存储接口（参见第2.8.4节） ◦ 2 禁用USB PICOBOT 接口（参见第2.8.5节）

'W','V'	<code>_wait_for_vector</code>
	This is the method that is entered by core 1 on reset to wait to be launched by core 0. There are few cases where you should call this method (resetting core 1 is much better). This method does not return and should only ever be called on core 1.
'E','C'	<code>deprecated</code>
	Do not use this function which may not be present.

2.8.3.2. Fast Floating Point Library

The Bootrom contains an optimized single-precision floating point implementation. Additionally V2 onwards also contain an optimized double-precision float point implementation. The function pointers for each precision are kept in a table structure found via the `rom_data_lookup` table (see [Section 2.8.3.3](#)).

2.8.3.2.1. Implementation Details

There is always a trade-off between speed and size. Whilst the overall goal for the floating-point routines is to achieve good performance within a small footprint, the emphasis is more on improved performance for the basic operations (add, subtract, multiply, divide and square root) and more on reduced footprint for the scientific functions (trigonometric functions, logarithms and exponentials).

The IEEE single- and double-precision data formats are used throughout, but in the interests of reducing code size, input denormals are treated as zero, input NaNs are treated as infinities, output denormals are flushed to zero, and output NaNs are rendered as infinities. Only the round-to-nearest, even-on-tie rounding mode is supported. Traps are not supported.

The five basic operations return results that are always correctly rounded.

The scientific functions always return results within 1 ULP (unit in last place) of the exact result. In many cases results are better.

The scientific functions are calculated using internal fixed-point representations so accuracy (as measured in ULP error rather than in absolute terms) is poorer in situations where converting the result back to floating point entails a large normalising shift. This occurs, for example, when calculating the sine of a value near a multiple of pi, the cosine of a value near an odd multiple of pi/2, or the logarithm of a value near 1. Accuracy of the tangent function is also poorer when the result is very large. Although covering these cases is possible, it would add considerably to the code footprint, and there are few types of program where accuracy in these situations is essential.

The sine, cosine and tangent functions also only operate correctly over a limited range: $-128 < x < +128$ for single-precision arguments x and $-1024 < x < +1024$ for double-precision x. This is to avoid the need to (at least in effect) store the value of pi to high precision within the code, and hence saves code space. Accurate range reduction over a wider range of arguments can be done externally to the library if required, but again there are few situations where this would be needed.

NOTE

The SDK cos/sin functions perform this range reduction, so accept the full range of arguments, though are slower for inputs outside of these ranges.

2.8.3.2.2. Functions

These functions follow the standard ARM EABI for passing floating point values.

You do not need to call these methods directly as the SDK `pico_float` and `pico_double` libraries used by default replace the ARM EABI *Float functions* such that C/C++ level code (or indirectly code in languages such as *MicroPython* that are implemented in C) use these Bootrom functions automatically for the corresponding floating point operations.

'W', 'V'	<code>_wait_for_vector</code> 此方法由核心1在复位时进入，等待核心0发起启动。极少数情况下应调用此方法（重置核心1更为合适）。此方法不会返回，且应仅在核心1上调用。
'E', 'C'	已废弃 请勿使用此函数，因其可能不存在。

2.8.3.2 快速浮点库

Bootrom 包含经过优化的单精度浮点实现。此外，从 V2 版本开始，还包含经过优化的双精度浮点实现。每种精度对应的函数指针保存在通过 `rom_data_lookup` 表查找的表结构中（详见第 2.8.3.3 节）。

2.8.3.2.1 实现细节

速度与体积之间始终存在权衡。尽管浮点例程的总体目标是在较小体积内实现优良性能，重点更侧重于基本运算（加、减、乘、除及平方根）的性能提升，以及科学函数（三角函数、对数及指数函数）的体积压缩。

整个过程中采用 IEEE 单精度和双精度数据格式，但为了减小代码体积，输入的非规格化数被视为零，输入的 NaN 被视为无穷大，输出的非规格化数被冲洗为零，输出的 NaN 被渲染为无穷大。仅支持四舍五入至最近且遇5取偶的舍入模式。不支持触发异常。

五种基本运算返回的结果始终正确舍入。

科学函数的返回结果始终在精确值的 1 ULP（单位最后位）误差范围内。在许多情况下，结果更为准确。

科学函数使用内部定点表示进行计算，因此在将结果转换回浮点数时，如需大量规整移位的情况下，准确度（以 ULP 误差衡量，而非绝对误差）较差。例如，计算接近 π 倍数的正弦值、接近 $\pi/2$ 奇数倍的余弦值或接近 1 的对数值时，会出现此情形。当结果极大时，正切函数的精度亦较差。虽然覆盖这些情况是可行的，但会显著增加代码体积，且极少有程序类型在此类情形下对精度有严格要求。

正弦、余弦及正切函数仅在有限范围内正确运行：单精度参数 x 的范围为 $-128 < x < +128$ ，双精度参数 x 的范围为 $-1024 < x < +1024$ 。此设计旨在避免代码中（至少实际效果上）存储高精度 π 值，因而节省代码空间。若需进行更广泛参数范围的准确约简，可在库外完成，但此类需求极为罕见。

① 注意

SDK 的 cos/sin 函数执行该范围约简，因此可接受全范围参数，但对于超出该范围的输入，运算速度较慢。

2.8.3.2.2. 函数

这些函数遵循标准 ARM EABI 的浮点数传递规范。

您无需直接调用这些方法，因为默认使用的 SDK `pico_float` 和 `pico_double` 库已替代 ARM EABI `Float` 函数，使得 C/C++ 级别代码（或间接受 C 语言实现的语言，如 `MicroPython`）自动调用这些 Bootrom 函数以执行相应的浮点运算。

Some of these functions do not behave exactly the same as some of the corresponding C library functions. For that reason if you are using the SDK it is strongly advised that you simply use the regular `math.h` functions or those in `pico/float.h` or `pico/double.h` and not try to call into the bootrom directly.

Note that double-precision floating point support is not present in version 1 (V1) of the bootrom, but the above mentioned `pico_double` library in the SDK will take care of pulling in any extra code needed for V1.

NOTE

For more information on using floating point in the SDK, and real world timings (noting also that some conversion functions are re-implemented in the SDK to be faster) see [floating point support](#).

Table 169. Single-precision Floating Point Function Table. Timings are average time in us over random (worst case) input. Functions with timing of N/A are not present in that ROM version, and the function pointer should be considered invalid. The functions (and table entries) from offset 0x54 onwards are only present in the V2 ROM.

Offset	V1 Cycles (Avg)	V2/V3 Cycles (Avg)	Description
Functions common to all versions of the bootrom			
0x00	71	71	<code>float _fadd(float a, float b)</code> Return a + b
0x04	74	74	<code>float _fsub(float a, float b)</code> Return a - b
0x08	69	58	<code>float _fmul(float a, float b)</code> Return a * b
0x0c	71	71	<code>float _fdiv(float a, float b)</code> Return a / b
0x10	N/A	N/A	deprecated Do not use this function
0x14	N/A	N/A	deprecated Do not use this function
0x18	63	63	<code>float _fsqrt(float v)</code> Return \sqrt{v} or $-\infty$ if v is negative. (Note V1 returns $+\infty$ in this case)
0x1c	37	40	<code>int _float2int(float v)</code> Convert a float to a signed integer, rounding towards $-\infty$, and clamping the result to lie within the range <code>-0x80000000</code> to <code>0x7FFFFFFF</code>
0x20	36	39	<code>int _float2fix(float v, int n)</code> Convert a float to a signed fixed point integer representation where n specifies the position of the binary point in the resulting fixed point representation - e.g. <code>_float2fix(0.5f, 16) == 0x8000</code> . This method rounds towards $-\infty$, and clamps the resulting integer to lie within the range <code>-0x80000000</code> to <code>0x7FFFFFFF</code>
0x24	38	39	<code>uint _float2uint(float v)</code> Convert a float to an unsigned integer, rounding towards $-\infty$, and clamping the result to lie within the range <code>0x00000000</code> to <code>0xFFFFFFFF</code>

某些函数的行为与对应的 C 标准库函数并不完全相同。因此，强烈建议在使用 SDK 时，直接使用常规的 `math.h` 函数或 `pico/float.h` 及 `pico/double.h` 中的函数，而非尝试直接调用 Bootrom。

请注意，Bootrom 第一版（V1）不支持双精度浮点，但 SDK 中上述的 `pico_double` 库会自动引入 V1 所需的额外代码。

注意

有关在 SDK 中使用浮点数及其实际运行时间的详细信息（同时注意部分转换函数已被 SDK 重新实现以提升速度），请参阅浮点支持章节。

表169。单精度浮点函数表。

时序为随机（最差情况）输入
入下的平均时间，单位为微秒。时序显示为 N/A 的函数在该 ROM 版本中不存在，函数指针应视为无效。偏移量 0x54 及之后的函数（及表项）仅存在于 V2 ROM 版本中。

偏移量	V1周期 (平均)	V2/V3 周期 (平均)	描述
所有引导 ROM 版本通用函数			
0x00	71	71	float _fadd(float a, float b) 返回 $a + b$
0x04	74	74	float _fsub(float a, float b) 返回 $a - b$
0x08	69	58	float _fmul(float a, float b) 返回 $a * b$
0x0c	71	71	float _fdiv(float a, float b) 返回 a / b
0x10	不适用	不适用	已废弃 请勿使用此函数
0x14	不适用	不适用	已废弃 请勿使用此函数
0x18	63	63	float _fsqrt(float v) 如果 v 为负数，则返回正无穷或负无穷。（注意 V1 在此情况下返回正无穷）
0x1c	37	40	int _float2int(float v) 将浮点数转换为有符号整数，向负无穷方向舍入，并将结果限制在范围 <code>-0x80000000</code> 至 <code>0x7FFFFFFF</code>
0x20	36	39	int _float2fix(float v, int n) 将浮点数转换为有符号定点整数表示，其中 n 指定二进制小数点在结果定点表示中的位置，例如 <code>_float2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 <code>-0x80000000</code> 至 <code>0x7FFFFFFF</code>
0x24	38	39	uint _float2uint(float v) 将浮点数转换为无符号整数，向负无穷方向舍入，并将结果限制在范围 <code>0x00000000</code> 至 <code>0xFFFFFFFF</code>

0x28	38	38	<code>uint _float2ufix(float v, int n)</code> Convert a float to an unsigned fixed point integer representation where n specifies the position of the binary point in the resulting fixed point representation, e.g. <code>_float2ufix(0.5f, 16) == 0x8000</code> . This method rounds towards <i>-Infinity</i> , and clamps the resulting integer to lie within the range <code>0x00000000</code> to <code>0xFFFFFFFF</code>
0x2c	55	55	<code>float _int2float(int v)</code> Convert a signed integer to the nearest float value, rounding to even on tie
0x30	53	53	<code>float _fix2float(int32_t v, int n)</code> Convert a signed fixed point integer representation to the nearest float value, rounding to even on tie. n specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x34	54	54	<code>float _uint2float(uint32_t v)</code> Convert an unsigned integer to the nearest float value, rounding to even on tie
0x38	52	52	<code>float _ufix2float(uint32_t v, int n)</code> Convert an unsigned fixed point integer representation to the nearest float value, rounding to even on tie. n specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x3c	603	587	<code>float _fcos(float angle)</code> Return the cosine of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -128 to 128
0x40	593	577	<code>float _fsin(float angle)</code> Return the sine of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -128 to 128
0x44	669	653	<code>float _ftan(float angle)</code> Return the tangent of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -128 to 128
0x48	N/A	N/A	deprecated Do not use this function
0x4c	542	524	<code>float _fexp(float v)</code> Return the exponential value of <i>v</i> , i.e. so e^v
0x50	810	789	<code>float _fln(float v)</code> Return the natural logarithm of <i>v</i> . If $v \leq 0$ return <i>-Infinity</i>
Functions (and table entries) present in the V2/V3 bootrom only			
0x54	N/A	25	<code>int _fcmp(float a, float b)</code> Compares two floating point numbers, returning: <ul style="list-style-type: none">• 0 if $a == b$• -1 if $a < b$• 1 if $a > b$
0x58	N/A	667	<code>float _fatan2(float y, float x)</code> Computes the arc tangent of y/x using the signs of arguments to determine the correct quadrant

0x28	38	38	<code>uint _float2ufix(float v, int n)</code> 将浮点数转换为无符号定点整数表示，其中 n 指定二进制小数点在结果定点表示中的位置，例如 <code>_float2ufix(0.5f, 16) == 0x8000</code> 。该方法向 $-\infty$ 方向舍入，且将结果整数限制在范围 0x00000000 至 0xFFFFFFFF
0x2c	55	55	<code>float _int2float(int v)</code> 将有符号整数转换为最接近的浮点值，平分情况采用偶数舍入
0x30	53	53	<code>float _fix2float(int32_t v, int n)</code> 将有符号定点整数表示转换为最接近的浮点值， 平分情况采用偶数舍入。 n 指定定点数中二进制小数点的位置， 因此 $= \text{nearest}(v / 2^n)$
0x34	54	54	<code>float _uint2float(uint32_t v)</code> 将无符号整数转换为最接近的浮点值，平分情况采用偶数舍入
0x38	52	52	<code>float _ufix2float(uint32_t v, int n)</code> 将无符号定点整数表示转换为最接近的浮点数值， 平分情况采用偶数舍入。 n 指定定点数中二进制小数点的位置， 因此 $= \text{nearest}(v / 2^n)$
0x3c	603	587	<code>float _fcos(float angle)</code> 返回 angle 的余弦值。 angle 的单位为弧度，且必须处于 -128 至 128 范围内
0x40	593	577	<code>float _fsin(float angle)</code> 返回 angle 的正弦值。 angle 的单位为弧度，且必须处于 -128 至 128 范围内
0x44	669	653	<code>float _ftan(float angle)</code> 返回 angle 的正切值。 angle 的单位为弧度，且必须处于 -128 至 128 范围内
0x48	不适用	不适用	已废弃 请勿使用此函数
0x4c	542	524	<code>float _fexp(float v)</code> 返回 v 的指数值，即 e 的 v 次幂 e^v
0x50	810	789	<code>float _fln(float v)</code> 返回 v 的自然对数。如果 $v <= 0$ 返回 -无穷大
仅存在于 V2/V3 启动只读存储器中的函数（及表项）			
0x54	不适用	25	<code>int _fcmp(float a, float b)</code> 比较两个浮点数，返回： <ul style="list-style-type: none">• 0，如果 $a == b$• -1，如果 $a < b$• 1，如果 $a > b$
0x58	不适用	667	<code>float _fatan2(float y, float x)</code> 计算 y/x 的反正切，通过参数符号确定正确象限

0x5c	N/A	62	<code>float _int642float(int64_t v)</code> Convert a signed 64-bit integer to the nearest float value, rounding to even on tie
0x60	N/A	60	<code>float _fix642float(int64_t v, int n)</code> Convert a signed fixed point 64-bit integer representation to the nearest float value, rounding to even on tie. n specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x64	N/A	58	<code>float _uint642float(uint64_t v)</code> Convert an unsigned 64-bit integer to the nearest float value, rounding to even on tie
0x68	N/A	57	<code>float _ufix642float(uint64_t v, int n)</code> Convert an unsigned fixed point 64-bit integer representation to the nearest float value, rounding to even on tie. n specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x6c	N/A	54	<code>_float2int64</code> Convert a float to a signed 64-bit integer, rounding towards <i>-Infinity</i> , and clamping the result to lie within the range <code>0x8000000000000000</code> to <code>0x7FFFFFFFFFFFFF</code>
0x70	N/A	53	<code>_float2fix64</code> Convert a float to a signed fixed point 64-bit integer representation where n specifies the position of the binary point in the resulting fixed point representation - e.g. <code>_float2fix(0.5f, 16) == 0x8000</code> . This method rounds towards <i>-Infinity</i> , and clamps the resulting integer to lie within the range <code>0x8000000000000000</code> to <code>0x7FFFFFFFFFFFFF</code>
0x74	N/A	42	<code>_float2uint64</code> Convert a float to an unsigned 64-bit integer, rounding towards <i>-Infinity</i> , and clamping the result to lie within the range <code>0x0000000000000000</code> to <code>0xFFFFFFFFFFFFFFF</code>
0x78	N/A	41	<code>_float2ufix64</code> Convert a float to an unsigned fixed point 64-bit integer representation where n specifies the position of the binary point in the resulting fixed point representation, e.g. <code>_float2ufix(0.5f, 16) == 0x8000</code> . This method rounds towards <i>-Infinity</i> , and clamps the resulting integer to lie within the range <code>0x0000000000000000</code> to <code>0xFFFFFFFFFFFFFFF</code>
0x7c	N/A	15	<code>double _float2double(float v)</code> Converts a float to a double
Function present in the V3 bootrom only			
0x48 (uses previously deprecated slot)	577 (V3 only)	577 (V3 only)	<code>float (float) _fsincos(float angle)</code> Calculates the sine and cosine of $angle$. $angle$ is in radians, and must be in the range -128 to 128. The sine value is returned in register $r0$ (and is thus the official function return value), the cosine value is returned in register $r1$. This method is considerably faster than calling <code>_fsin</code> and <code>_fcos</code> separately.

Note that the V2/V3 bootroms contains an equivalent table of functions for double-precision floating point operations. The offsets are the same, however where there was now float there is double (and vice versa for the float<>double conversion)

0x5c	不适用	62	<code>float _int642float(int64_t v)</code> 将有符号64位整数转换为最接近的浮点数值，平分时采用偶数舍入法
0x60	不适用	60	<code>float _fix642float(int64_t v, int n)</code> 将有符号定点64位整数表示转换为最接近的浮点数值，平分时采用偶数舍入法。n指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x64	不适用	58	<code>float _uint642float(uint64_t v)</code> 将无符号64位整数转换为最接近的浮点值，平局时四舍六入
0x68	不适用	57	<code>float _ufix642float(uint64_t v, int n)</code> 将无符号定点64位整数表示转换为最接近的浮点值，平局时四舍六入。n指定定点数中二进制点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x6c	不适用	54	<code>_float2int64</code> 将浮点数转换为有符号64位整数，采用向负无穷舍入，并将结果限制在范围-0x8000000000000000至0x7FFFFFFFFFFFFF
0x70	不适用	53	<code>_float2fix64</code> 将浮点数转换为有符号定点64位整数表示，其中n指定结果定点表示的二进制点位置 - 例如： <code>_float2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷舍入，且将结果整数限定在范围-0x8000000000000000至0x7FFFFFFFFFFFFF
0x74	不适用	42	<code>_float2uint64</code> 将浮点数转换为无符号64位整数，向-∞舍入，并将结果限制在范围0x0000000000000000至0xFFFFFFFFFFFFFF
0x78	不适用	41	<code>_float2ufix64</code> 将浮点数转换为无符号定点64位整数表示，其中n指定结果定点表示中二进制点的位置， 例如 <code>_float2ufix(0.5f, 16) == 0x8000</code> 。此方法向-∞舍入，并将结果整数限制在范围0x0000000000000000至0xFFFFFFFFFFFFFF
0x7c	不适用	15	<code>double _float2double(float v)</code> 将浮点数转换为双精度浮点数
此函数仅存在于V3启动只读存储器中			
0x48 (使用先前已弃用的槽位)	<small>577 (仅适用于V3)</small>		<code>float (float) _fsincos(float 角度)</code>
			计算指定角度的正弦和余弦。角度以弧度表示，且必须位于-128至128范围内。 正弦值存储于寄存器r0（因此为官方函数返回值），余弦值存储于寄存器r1。该方法比分别调用 <code>_fsin</code> 和 <code>_fcos</code> 显著更快。

请注意，V2/V3启动ROM包含等效的双精度浮点运算函数表。

偏移量相同，但原为float的位置改为double（反之亦然，用于float<>double转换）。

Table 170. Double-precision Floating Point Function Table.
Timings are average time in us over random (worst case) input. Functions with timing of N/A are not present in that ROM version, and the function pointer should be considered invalid. The functions (and table entries) from offset 0x54 onwards are only present in the V2 and V3 ROMs.

Offset	Cycles (Avg)*	Description
0x00	91	<code>double _dadd(double a, double b)</code>
		Return $a + b$
0x04	95	<code>double _dsub(double a, double b)</code>
		Return $a - b$
0x08	155	<code>double _dmul(double a, double b)</code>
		Return $a * b$
0x0c	183	<code>double _ddiv(double a, double b)</code>
		Return a / b
0x10	N/A	deprecated
		Do not use this function
0x14	N/A	deprecated
		Do not use this function
0x18	169	<code>double _dsqrt(double v)</code>
		Return \sqrt{v} or $-\text{Infinity}$ if v is negative.
0x1c	75	<code>int _double2int(double v)</code>
		Convert a double to a signed integer, rounding towards $-\text{Infinity}$, and clamping the result to lie within the range <code>-0x80000000</code> to <code>0x7FFFFFFF</code>
0x20	74	<code>int _double2fix(double v, int n)</code>
		Convert a double to a signed fixed point integer representation where n specifies the position of the binary point in the resulting fixed point representation - e.g. <code>_double2fix(0.5f, 16) == 0x8000</code> . This method rounds towards $-\text{Infinity}$, and clamps the resulting integer to lie within the range <code>-0x80000000</code> to <code>0x7FFFFFFF</code>
0x24	63	<code>uint _double2uint(double v)</code>
		Convert a double to an unsigned integer, rounding towards $-\text{Infinity}$, and clamping the result to lie within the range <code>0x00000000</code> to <code>0xFFFFFFFF</code>
0x28	62	<code>uint _double2ufix(double v, int n)</code>
		Convert a double to an unsigned fixed point integer representation where n specifies the position of the binary point in the resulting fixed point representation, e.g. <code>_double2ufix(0.5f, 16) == 0x8000</code> . This method rounds towards $-\text{Infinity}$, and clamps the resulting integer to lie within the range <code>0x00000000</code> to <code>0xFFFFFFFF</code>
0x2c	69	<code>double _int2double(int v)</code>
		Convert a signed integer to the nearest double value, rounding to even on tie
0x30	68	<code>double _fix2double(int32_t v, int n)</code>
		Convert a signed fixed point integer representation to the nearest double value, rounding to even on tie. n specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x34	64	<code>double _uint2double(uint32_t v)</code>
		Convert an unsigned integer to the nearest double value, rounding to even on tie

表 170。双精度浮点函数表。

时序为随机（最差情况）输入下的平均时间，单位为微秒。时序显示为N/A的函数在该ROM版本中不存在，函数指针应视为无效。偏移量自0x54起的函数及表项仅存在于V2和V3 ROM中。

偏移量	周期数 (平均值) *	描述
0x00	91	<code>double _dadd(double a, double b)</code> 返回 $a + b$
0x04	95	<code>double _dsub(double a, double b)</code> 返回 $a - b$
0x08	155	<code>double _dmul(double a, double b)</code> 返回 $a * b$
0x0c	183	<code>double _ddiv(double a, double b)</code> 返回 a / b
0x10	不适用	已废弃 请勿使用此函数
0x14	不适用	已废弃 请勿使用此函数
0x18	169	<code>double _dsqrt(double v)</code> 如果 v 为负，则返回 0 或 $-\infty$ 。
0x1c	75	<code>int _double2int(double v)</code> 将 double 转换为有符号整数，向 $-\infty$ 舍入，并将结果限制在范围 $0x8000000$ 至 $0x7FFFFFFF$
0x20	74	<code>int _double2fix(double v, int n)</code> 将 double 转换为有符号定点整数表示，其中 n 指定结果中二进制小数点的位置——例如 <code>_double2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 $0x8000000$ 至 $0x7FFFFFFF$
0x24	63	<code>uint _double2uint(double v)</code> 将 double 转换为无符号整数，向 $-\infty$ 舍入，并将结果限制在范围 $0x0$ 至 $0xFFFFFFFF$
0x28	62	<code>uint _double2ufix(double v, int n)</code> 将 double 转换为无符号定点整数表示，其中 n 指定结果中二进制小数点的位置，例如 <code>_double2ufix(0.5f, 16) == 0x8000</code> 。该方法向 $-\infty$ 方向舍入，且将结果整数限制在范围 $0x00000000$ 至 $0xFFFFFFFF$
0x2c	69	<code>double _int2double(int v)</code> 将有符号整数转换为最接近的 double 值，平分时向偶数舍入
0x30	68	<code>double _fix2double(int32_t v, int n)</code> 将有符号定点整数表示转换为最接近的 double 值，平分时向偶数舍入。 n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$
0x34	64	<code>double _uint2double(uint32_t v)</code> 将无符号整数转换为最接近的 double 值，平分时向偶数舍入

Offset	Cycles (Avg)*	Description
0x38	62	<code>double _ufix2double(uint32_t v, int n)</code> Convert an unsigned fixed point integer representation to the nearest double value, rounding to even on tie. <i>n</i> specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x3c	1617	<code>double _dcos(double angle)</code> Return the cosine of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -1024 to 1024
0x40	1618	<code>double _dsin(double angle)</code> Return the sine of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -1024 to 1024
0x44	1891	<code>double _dtan(double angle)</code> Return the tangent of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -1024 to 1024
0x48	N/A	deprecated Do not use this function
0x4c	804	<code>double _dexp(double v)</code> Return the exponential value of <i>v</i> , i.e. so e^v
0x50	428	<code>double _dln(double v)</code> Return the natural logarithm of <i>v</i> . If $v \leq 0$ return <i>-Infinity</i>
0x54	39	<code>int _dcmp(double a, double b)</code> Compares two floating point numbers, returning: <ul style="list-style-type: none">• 0 if <i>a</i> == <i>b</i>• -1 if <i>a</i> < <i>b</i>• 1 if <i>a</i> > <i>b</i>
0x58	2168	<code>double _datan2(double y, double x)</code> Computes the arc tangent of <i>y/x</i> using the signs of arguments to determine the correct quadrant
0x5c	55	<code>double _int642double(int64_t v)</code> Convert a signed 64-bit integer to the nearest double value, rounding to even on tie
0x60	56	<code>double _dix642double(int64_t v, int n)</code> Convert a signed fixed point 64-bit integer representation to the nearest double value, rounding to even on tie. <i>n</i> specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x64	50	<code>double _uint642double(uint64_t v)</code> Convert an unsigned 64-bit integer to the nearest double value, rounding to even on tie
0x68	49	<code>double _ufix642double(uint64_t v, int n)</code> Convert an unsigned fixed point 64-bit integer representation to the nearest double value, rounding to even on tie. <i>n</i> specifies the position of the binary point in fixed point, so $f = \text{nearest}(v / 2^n)$
0x6c	64	<code>_double2int64</code> Convert a double to a signed 64-bit integer, rounding towards <i>-Infinity</i> , and clamping the result to lie within the range <code>-0x8000000000000000</code> to <code>0x7FFFFFFFFFFFFFFFFF</code>

偏移量	周期数 (平均值) *	描述
0x38	62	<p><code>double _ufix2double(uint32_t v, int n)</code></p> <p>将无符号定点整数表示转换为最接近的 double 值，平分时向偶数舍入。n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x3c	1617	<p><code>double _dcos(double angle)</code></p> <p>返回 angle 的余弦值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x40	1618	<p><code>double _dsin(double angle)</code></p> <p>返回 angle 的正弦值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x44	1891	<p><code>double _dtan(double angle)</code></p> <p>返回 angle 的正切值。angle 以弧度为单位，且必须在 -1024 至 1024 范围内</p>
0x48	不适用	<p>已废弃</p> <p>请勿使用此函数</p>
0x4c	804	<p><code>double _dexp(double v)</code></p> <p>返回 v 的指数值，即 e 的 v 次幂 e^v</p>
0x50	428	<p><code>double _dln(double v)</code></p> <p>返回 v 的自然对数。如果 $v \leq 0$ 返回 -无穷大</p>
0x54	39	<p><code>int _dcmp(double a, double b)</code></p> <p>比较两个浮点数，返回：</p> <ul style="list-style-type: none"> • 0，如果 $a == b$ • -1，如果 $a < b$ • 1，如果 $a > b$
0x58	2168	<p><code>double _datan2(double y, double x)</code></p> <p>计算 y/x 的反正切，通过参数符号确定正确象限</p>
0x5c	55	<p><code>double _int642double(int64_t v)</code></p> <p>将有符号64位整数转换为最接近的double值，遇平局时向偶数舍入</p>
0x60	56	<p><code>double _dix642double(int64_t v, int n)</code></p> <p>将有符号64位整数表示转换为最接近的double值，遇平局时向偶数舍入。n指定定点表示中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x64	50	<p><code>double _uint642double(uint64_t v)</code></p> <p>将无符号64位整数转换为最接近的double值，遇平局时向偶数舍入</p>
0x68	49	<p><code>double _ufix642double(uint64_t v, int n)</code></p> <p>将无符号定点64位整数表示转换为最接近的double值，平分时向偶数舍入。n 指定定点数中二进制小数点的位置，因此 $f = \text{nearest}(v / 2^n)$</p>
0x6c	64	<p><code>_double2int64</code></p> <p>将double转换为有符号64位整数，向负无穷大舍入，并将结果限定在范围-0x8内0000000000000000至0x7FFFFFFFFFFFFF</p>

Offset	Cycles (Avg)*	Description
0x70	63	<code>_double2fix64</code> Convert a double to a signed fixed point 64-bit integer representation where <i>n</i> specifies the position of the binary point in the resulting fixed point representation - e.g. <code>_double2fix(0.5f, 16) == 0x8000</code> . This method rounds towards <i>-Infinity</i> , and clamps the resulting integer to lie within the range <code>-0x8000000000000000</code> to <code>0x7FFFFFFFFFFFFFFFFF</code>
0x74	53	<code>_double2uint64</code> Convert a double to an unsigned 64-bit integer, rounding towards <i>-Infinity</i> , and clamping the result to lie within the range <code>0x0000000000000000</code> to <code>0xFFFFFFFFFFFFFFFFF</code>
0x78	52	<code>_double2ufix64</code> Convert a double to an unsigned fixed point 64-bit integer representation where <i>n</i> specifies the position of the binary point in the resulting fixed point representation, e.g. <code>_double2ufix(0.5f, 16) == 0x8000</code> . This method rounds towards <i>-Infinity</i> , and clamps the resulting integer to lie within the range <code>0x0000000000000000</code> to <code>0xFFFFFFFFFFFFFFFFF</code>
0x7c	23	<code>float _double2float(double v)</code> Converts a double to a float
Function present in the V3 bootrom only		
0x48	1718 (uses previously deprecated slot)	<code>double (,double) _sincos(double angle)</code> Calculates the sine and cosine of <i>angle</i> . <i>angle</i> is in radians, and must be in the range -1024 to 1024. The sine value is returned in registers <i>r0/r1</i> (and is thus the official return value), the cosine value is returned in registers <i>r2/r3</i> . This method is considerably faster than calling <code>_sin</code> and <code>_cos</code> separately.

2.8.3.3. Bootrom Data

The Bootrom data table (`rom_data_table`) contains the following pointers.

Table 171. Bootrom data pointers

CODE	Value (16-bit pointer) Description
'C', 'R'	<code>const char *copyright_string</code> The Raspberry Pi Trading Ltd copyright string.
'G', 'R'	<code>const uint32_t *git_revision</code> The 8 most significant hex digits of the Bootrom git revision.
'F', 'S'	<code>fplib_start</code> The start address of the floating point library code and data. This and <code>fplib_end</code> along with the individual function pointers in <code>soft_float_table</code> can be used to copy the floating point implementation into RAM if desired.
'S', 'F'	<code>soft_float_table</code> See Table 169 for the contents of this table.
'F', 'E'	<code>fplib_end</code> The end address of the floating point library code and data.

偏移量	周期数 (平均值) *	描述
0x70	63	<code>_double2fix64</code> 将 double 转换为有符号定点 64 位整数表示， <i>n</i> 指定结果定点表示中二进制小数点的位置——例如： <code>_double2fix(0.5f, 16) == 0x8000</code> 。该方法向负无穷方向舍入，并将结果整数限制在范围 0x8000000000000000 至 0x7FFFFFFFFFFFFF
0x74	53	<code>_double2uint64</code> 将 double 转换为无符号 64 位整数，向 -∞ 舍入，并将结果限制在 0x 范围内 0000000000000000 至 0xFFFFFFFFFFFFFF
0x78	52	<code>_double2ufix64</code> 将 double 转换为无符号定点 64 位整数表示，其中 <i>n</i> 指定二进制点在最终定点表示中的位置，例如 <code>_double2ufix(0.5f, 16) == 0x8000</code> 。该方法向 -∞ 方向舍入，且将结果整数限制在范围 0x0000000000000000 至 0xFFFFFFFFFFFFFF
0x7c	23	<code>float _double2float(double v)</code> 将 double 转换为 float
此函数仅存在于 V3 启动只读存储器中		
0x48 (使用先前已弃用的槽位)	1718 (仅限 V3)	<code>double (,double) _sincos(double angle)</code> 计算指定角度的正弦和余弦。 <i>angle</i> 单位为弧度，且必须位于 -1024 至 1024 范围内。正弦值返回至寄存器 <i>r0/r1</i> （因此为官方返回值），余弦值返回至寄存器 <i>r2/r3</i> 。此方法明显快于分别调用 <code>_sin</code> 和 <code>_cos</code> 。

2.8.3.3. Bootrom 数据

Bootrom 数据表 (`rom_data_table`) 包含以下指针。

表 171. Bootrom 数据指针

代码	值 (16 位指针) 描述
'C', 'R'	<code>const char *copyright_string</code> Raspberry Pi Trading Ltd 的版权声明字符串。
'G', 'R'	<code>const uint32_t *git_revision</code> Bootrom git 修订版的最高 8 位十六进制数字。
'F', 'S'	<code>fplib_start</code> 浮点库代码和数据的起始地址。该地址与 <code>fplib_end</code> 及 <code>soft_float_table</code> 中的各函数指针一起，可用于将浮点实现复制至 RAM（如需）。
'S', 'F'	<code>soft_float_table</code> 有关本表内容，详见表 169。
'F', 'E'	<code>fplib_end</code> 浮点库代码和数据的结束地址。

'S','D'	<code>soft_double_table</code>
This entry is only present in the V2 bootrom. See Table 170 for the contents of this table.	
'P','8'	<i>deprecated.</i> This entry is not present in the V2 bootrom; do not use it.
'R','8'	<i>deprecated.</i> This entry is not present in the V2 bootrom; do not use it.
'L','8'	<i>deprecated.</i> This entry is not present in the V2 bootrom; do not use it.
'T','8'	<i>deprecated.</i> This entry is not present in the V2 bootrom; do not use it.

2.8.4. USB Mass Storage Interface

The Bootrom provides a standard USB bootloader that makes a writeable drive available for copying code to the RP2040 using *UF2* files (see [Section 2.8.4.2](#)).

A *UF2* file copied to the drive is downloaded and written to Flash or RAM, and the device is automatically rebooted, making it trivial to download and run code on the RP2040 using only a USB connection.

2.8.4.1. The RPI-RP2 Drive

The RP2040 appears as a standard 128MB flash drive named *RPI-RP2* formatted as a single partition with FAT16. There are only ever two actual files visible on the drive specified.

- `INFO_UF2.TXT` - contains a string description of the UF2 bootloader and version.
- `INDEX.HTM` - redirects to information about the RP2040 device.

Any type of files may be written to the USB drive from the host, however in general these are not stored, and only appear to be so because of caching on the host side.

When a *UF2* file is written to the device however, the special contents are recognized and data is written to specified locations in RAM or Flash. On the completed download of an entire valid *UF2* file, the RP2040 automatically reboots to run the newly downloaded code.

NOTE

The `INDEX.HTM` file is currently redirected to <https://www.raspberrypi.com/documentation/microcontrollers/>

2.8.4.2. UF2 Format Details

TIP

To generate *UF2* files, use the *UF2* convert functionality in `picotool`.

NOTE

Invalid *UF2* files may not write at all or only write partially to RP2040 before failing. Not all operating systems notify you of disk write errors after a failed write. You can use `picotool` to verify that a *UF2* file wrote correctly to RP2040.

- All data destined for the device must be in a *UF2* block with `familyID` present and set to `0xe48bff56`, and a `payload_size` of `256`.
- All data must be destined for (and fit entirely within) the following memory ranges (depending on the type of binary being downloaded which is determined by the address of the first *UF2* block encountered):

'S','D'	<code>soft_double_table</code>
此条目仅存在于 V2 版本 bootrom 中。有关本表内容, 请参见表170。	
'P','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'R','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'L','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。
'T','8'	已弃用。该条目在V2引导ROM中不存在; 请勿使用。

2.8.4. USB大容量存储设备接口

引导ROM提供了标准USB引导加载程序, 使得可通过 *UF2*文件 (见第2.8.4.2节) 将代码复制到RP2040的可写驱动器。

复制到驱动器的 *UF2*文件将被下载并写入闪存或RAM, 设备自动重启, 从而仅凭USB连接即可轻松下载并运行RP2040上的代码。

2.8.4.1. RPI-RP2驱动器

RP2040以名为 *RPI-RP2*的标准128MB闪存驱动器形式出现, 格式为单一FAT16分区。驱动器上始终仅显示两个实际文件。

- `INFO_UF2.TXT` — 包含UF2引导加载程序及版本的字符串描述。
- `INDEX.HTM` - 重定向至关于RP2040设备的信息。

主机可以向USB驱动器写入任何类型的文件, 但通常这些文件不会被实际存储, 仅因主机端缓存而呈现为已存储状态。

当向设备写入 *UF2*文件时, 设备会识别其特殊内容, 并将数据写入RAM或Flash的指定位置。在整个有效 *UF2*文件下载完成后, RP2040会自动重启以运行新下载的代码。

ⓘ 注意

当前 `INDEX.HTM`文件重定向至 <https://www.raspberrypi.com/documentation/microcontrollers/>

2.8.4.2. UF2格式详情

💡 提示

要生成 *UF2*文件, 请使用picotool中的 *UF2*转换功能。

ⓘ 注意

无效的 *UF2*文件可能无法写入, 或仅部分写入RP2040后即失败。并非所有操作系统都会在写入失败后通知磁盘写入错误。您可以使用 `picotool`验证 *UF2*文件是否已正确写入RP2040。

- 发送至设备的所有数据必须包含在带有familyID字段且该字段值设为0xe48bff56的UF2数据块内, 且payload_size须为256。
- 所有数据必须定位在 (且完全位于) 以下内存范围内, 具体取决于第一个遇到的 *UF2*数据块地址, 该地址决定所下载二进制文件的类型:

a. A regular flash binary

- **0x10000000-0x11000000 Flash:** All blocks must be targeted at 256 byte alignments. Writes beyond the end of physical flash will wrap back to the beginning of flash.

b. A RAM only binary

- **0x20000000-0x20042000 Main RAM:** Blocks can be positioned with byte alignment.
- **0x15000000-0x15004000 Flash Cache:** (since flash is not being targeted, the Flash Cache is available for use as RAM with same properties as *Main RAM*).

NOTE

Traditionally UF2 has only been used to write to Flash, but this is more a limitation of using the metadata-free .BIN file as the source to generate the UF2 file. RP2040 takes full advantage of the inherent flexibility of UF2 to support the full range of binaries in the richer .ELF format produced by the build to be used as the source for the UF2 file.

- The `numBlocks` must specify a total size of the binary that fits in the regions specified above
- A change of `numBlocks` or the binary type (determined by UF2 block target address) will discard the current transfer in progress.
- All data must be in blocks without the `UF2_FLAG_NOT_MAIN_FLASH` marking which relates to content to be ignored rather than Flash vs RAM.

The flash is always erased a 4kB sector at a time, so including data for only a subset of the 256-byte pages within a sector in a flash-binary UF2 will leave the remaining 256-byte pages of the sector erased but undefined. The RP2040 bootrom will accept UF2 binaries with such partially-filled sectors, however due to a bug ([RP2040-E14](#)) such binaries may not be written correctly if there is any partially-filled sector other than at the end. Most flash binaries are 4kB aligned and contiguous, and therefore it is usually only the last sector that is partially-filled. If you need to write non-aligned or non-contiguous UF2s to flash, then you should make sure to include a full 4kB worth of data for every sector in flash that will be written other than the last. This is handled for you automatically by the `elf2uf2` tool in the SDK version 1.3.1 onwards, which explicitly adds zero-filled pages to the appropriate *partially-filled* sectors.

A binary is considered "downloaded" when each of the `numBlocks` blocks has been seen at least once in the course of a single valid transfer. The data for a block is only written the first time in case of the host resending duplicate blocks.

After downloading a regular flash binary, a reset is performed after which the flash binary second stage (at address `0x10000000` - the start of flash) will be entered (if valid) via the bootrom.

A downloaded RAM only binary is entered by watchdog reset into the start of the binary, which is calculated as the lowest address of a downloaded block (with Main RAM considered lower than Flash Cache if both are present).

Finally it is possible for host software to temporarily disable UF2 writes via the PICOBOOT interface to prevent interference with operations being performed via that interface (see below), in which case any UF2 file write in progress will be aborted.

2.8.5. USB PICOBOOT Interface

The PICOBOOT interface is a low level USB protocol for interacting with the RP2040 while it is in BOOTSEL mode. This interface may be used concurrently with the USB Mass Storage Interface.

It provides for flexible reading from and writing to RAM or Flash, rebooting, executing code on the device and a handful of other management functions.

Constants and structures related to the interface can be found in the SDK header https://github.com/raspberrypi/pico-sdk/blob/master/src/common/boot_picoboot_headers/include/boot/picoboot.h

a. 常规闪存二进制文件

- **0x10000000-0x11000000**闪存：所有数据块必须按照256字节对齐。写入超出物理闪存末端的部分将回绕至闪存起始位置。

b. 仅限RAM的二进制文件

- **0x20000000-0x20042000**主RAM：数据块可按字节对齐任意定位。
- **0x15000000-0x15004000**闪存缓存：鉴于未定位闪存，闪存缓存可用作具备与主RAM相同属性的RAM。

i 注意

传统上，UF2仅用于写入Flash，但这主要是由于采用无元数据的.BIN文件作为生成UF2文件源的限制。RP2040充分利用了UF2的固有灵活性，支持构建生成的更丰富的.ELF格式的完整二进制文件系列，作为生成UF2文件的源文件。

- numBlocks必须指定一个符合上述区域大小限制的二进制文件总大小。
- 更改 numBlocks或二进制类型（由UF2块的目标地址决定）将放弃当前正在进行的传输。
- 所有数据必须包含在未标记UF2_FLAG_NOT_MAIN_FLASH的区块中，该标记指示应忽略的内容，而非Flash与RAM的区分。

Flash始终以4kB扇区为单位擦除，因此，在包含Flash二进制UF2中某一扇区内部分256字节页的数据时，该扇区剩余的256字节页面将被擦除，但其内容未定义。RP2040启动只读存储器将接受包含此类部分填充扇区的UF2二进制文件，但因一个缺陷（RP2040-E14），若存在除末尾以外的任何部分填充扇区，可能导致此类二进制文件无法正确写入。大多数闪存二进制文件均为4kB对齐且连续，因此通常只有最后一个扇区是部分填充的。若需向闪存写入非对齐或非连续的UF2文件，应确保除最后一个扇区外，所有将写入的扇区均包含完整的4kB数据。从SDK版本1.3.1起，elf2uf2工具会自动处理此问题，明确向适当的部分填充扇区添加填充零页。

当在一次有效传输过程中，numBlocks个区块中的每一个至少被接收一次时，该二进制文件即被视为“已下载”。块的数据仅在主机重传重复块时首次写入。

下载常规闪存二进制文件后，将执行复位，随后通过bootrom进入闪存二进制第二阶段（地址0x1000000-闪存起始地址）（如有效）。

下载的仅RAM二进制文件通过看门狗复位进入二进制起始地址，该地址被计算为下载块中最低的地址（当主RAM和闪存缓存共存时，主RAM地址视为较低地址）。

主机软件可以通过PICOBLOCK接口临时禁用UF2写入操作，以防止对该接口正在执行的操作产生干扰（详见下文），在此过程中，任何进行中的UF2文件写入将被中止。

2.8.5. USB PICOBLOCK接口

PICOBLOCK接口是一种在RP2040处于BOOTSEL模式时用于交互的低级USB协议。该接口可与USB大容量存储接口同时使用。

该功能支持灵活地对RAM或Flash进行读写、设备重启、代码执行及多项其他管理操作。

接口相关的常量及结构定义载于SDK头文件：https://github.com/raspberrypi/pico-sdk/blob/master/src/common/boot_picoblock_headers/include/boot/picoblock.h

2.8.5.1. Identifying The Device

A RP2040 device is recognized by the *Vendor ID* and *Product ID* in its device descriptor (shown in [Table 172](#)).

Table 172. RP2040 Boot Device Descriptor

Field	Value
bLength	18
bDescriptorType	1
bcdUSB	1.10
bDeviceClass	0
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
idVendor	0x2e8a
idProduct	0x0003
bcdDevice	1.00
iManufacturer	1
iProduct	2
iSerial	3
bNumConfigurations	1

2.8.5.2. Identifying The Interface

The PICOBOOT interface is recognized by the "Vendor Specific" *Interface Class* and the zero *Interface Sub Class* and *Interface Protocol* (shown in [Table 173](#)). Note that you should not rely on the interface number, as that is dependent on whether the device is also exposing the Mass Storage Interface. Note also that the device equally may not be exposing the PICOBOOT interface at all, so you should not assume it is present.

Table 173. PICOBOOT Interface Descriptor

Field	Value
bLength	9
bDescriptorType	4
bInterfaceNumber	varies
bAlternateSetting	0
bNumEndpoints	2
bInterfaceClass	0xff (vendor specific)
bInterfaceSubClass	0
bInterfaceProtocol	0
ilInterface	0

2.8.5.3. Identifying The Endpoints

The PICOBOOT interface provides a single *BULK OUT* and a single *BULK IN* endpoint. These can be identified by their direction and type. You should not rely on endpoint numbers.

2.8.5.1. 设备识别

RP2040设备通过其设备描述符中的*Vendor ID*和*Product ID*进行识别（详见表172）。

表172。RP2040
启动设备
描述符

字段	数值
bLength	18
bDescriptorType	1
bcdUSB	1.10
bDeviceClass	0
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
idVendor	0x2e8a
idProduct	0x0003
bcdDevice	1.00
iManufacturer	1
iProduct	2
iSerial	3
bNumConfigurations	1

2.8.5.2 识别接口

PICOBOOT接口由「厂商专用（Vendor Specific）」接口类、零接口子类及接口协议识别（见表173）。请注意，不应依赖接口编号，该编号取决于设备是否同时暴露大容量存储接口。还应注意，设备可能根本未暴露PICOBOOT接口，因此不得假设其存在。

表173 PICOBOOT
接口描述符

字段	数值
bLength	9
bDescriptorType	4
bInterfaceNumber	可变
bAlternateSetting	0
bNumEndpoints	2
bInterfaceClass	0xff（厂商专用）
bInterfaceSubClass	0
bInterfaceProtocol	0
ilInterface	0

2.8.5.3. 识别端点

PICOBOOT接口提供一个单独的*BULK OUT*端点和一个单独的*BULK IN*端点。这些端点可根据其方向和类型进行识别。不应依赖端点编号。

2.8.5.4. PICOBLOCK Commands

The two bulk endpoints are used for sending commands and retrieving successful command results. All commands are exactly 32 bytes (see [Table 174](#)) and sent to the *BULK OUT* endpoint.

Table 174. PICOBLOCK Command Definition

Offset	Name	Description
0x00	dMagic	The value <code>0x431fd10b</code>
0x04	dToken	A user provided token to identify this request by
0x08	bCmdId	The ID of the command. Note that the top bit indicates data transfer direction (0x80 = IN)
0x09	bCmdSize	Number of bytes of valid data in the args field
0x0a	reserved	0x0000
0x0c	dTransferLength	The number of bytes the host expects to send or receive over the bulk channel
0x10	args	16 bytes of command specific data padded with zeros

If a command sent is invalid or not recognized, the bulk endpoints will be stalled. Further information will be available via the *GET_COMMAND_STATUS* request (see [Section 2.8.5.5.2](#)).

Following the initial 32 byte packet, if *dTransferLength* is non-zero, then many bytes are transferred over the bulk pipe and the command is completed with an empty packet in the opposite direction. If *dTransferLength* is zero then command success is indicated by an empty IN packet.

The following commands are supported (note common fields *dMagic*, *dToken*, *reserved* are omitted for clarity)

2.8.5.4.1. EXCLUSIVE_ACCESS (0x01)

Claim or release exclusive access for writing to the RP2040 over USB (versus the Mass Storage Interface)

Table 175. PICOBLOCK Exclusive access command structure

Offset	Name	Value / Description	
0x08	bCmdId	0x01 (EXCLUSIVE_ACCESS)	
0x09	bCmdSize	0x01	
0x0c	dTransferLength	0x00000000	
0x10	bExclusive	NOT_EXCLUSIVE (0)	No restriction on USB Mass Storage operation
		EXCLUSIVE (1)	Disable USB Mass Storage writes (the host should see them as write protect failures, but in any case any active UF2 download will be aborted)
		EXCLUSIVE_AND_EJECT (2)	Lock the USB Mass Storage Interface out by marking the drive media as not present (ejecting the drive)

2.8.5.4.2. REBOOT (0x02)

Reboots the RP2040 out of BOOTSEL mode. Note that BOOTSEL mode might be re-entered if rebooting to flash and no valid second stage bootloader is found.

Table 176. PICOBLOCK Reboot access command structure

Offset	Name	Value / Description
0x08	bCmdId	0x02 (REBOOT)
0x09	bCmdSize	0x0c

2.8.5.4. PICOBEST命令

两个bulk端点用于发送命令并检索成功的命令结果。所有命令均为32字节（参见表174），并通过BULK OUT端点发送。

表174。PICOBEST命令定义

偏移量	名称	描述
0x00	dMagic	值 0x431fd10b
0x04	dToken	用户提供的令牌，用于标识该请求
0x08	bCmdId	命令ID。注意，最高位表示数据传输方向（0x80 = IN）
0x09	bCmdSize	args字段中有效数据的字节数
0x0a	保留	0x0000
0x0c	dTransferLength	主机预期通过批量通道发送或接收的字节数
0x10	args	16字节特定命令数据，以零填充

若发送的命令无效或无法识别，批量端点将被阻塞。更多信息可通过GET_COMMAND_STATUS请求获得（见第2.8.5.5.2节）。

在初始32字节数据包之后，若dTransferLength非零，则通过批量传输管道传输相应字节数的数据，命令随后以相反方向的空数据包完成。若dTransferLength为零，则通过空的IN数据包表示命令成功。

支持以下命令（为简洁起见，省略公共字段dMagic、dToken及reserved）

2.8.5.4.1. EXCLUSIVE_ACCESS (0x01)

声明或释放通过USB对RP2040进行写入的排他访问权限（区别于大容量存储接口）

表175。PICOBEST排他访问命令结构

偏移量	名称	数值 / 描述	
0x08	bCmdId	0x01 (EXCLUSIVE_ACCESS)	
0x09	bCmdSize	0x01	
0x0c	dTransferLength	0x00000000	
0x10	bExclusive	NOT_EXCLUSIVE (0)	对USB大容量存储操作无限制
		EXCLUSIVE (1)	禁用USB大容量存储写入（主机应将其视为写保护失败，但任何活动的UF2下载将被中止）
		EXCLUSIVE_AND_EJECT (2)	通过标记驱动器媒体为不存在（弹出驱动器）来锁定USB大容量存储接口

2.8.5.4.2. REBOOT (0x02)

使RP2040从BOOTSEL模式重启。注意，若重启至闪存且未找到有效的第二阶段引导程序，可能会重新进入BOOTSEL模式。

表176。PICOBEST重启访问命令结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x02（重启）
0x09	bCmdSize	0x0c

0x0c	dTransferLength	0x00000000	
0x10	dPC	The address to start executing from. Valid values are: 0x00000000	
		RAM address	Reboot via the standard Flash boot mechanism Reboot via watchdog and start executing at the specified address in RAM
0x14	dSP	Initial stack pointer post reboot (only used if booting into RAM)	
0x18	dDelayMS	Number of milliseconds to delay prior to reboot	

2.8.5.4.3. FLASH_ERASE (0x03)

Erases a contiguous range of flash sectors.

Table 177. PICOBEST
Flash erase command structure

Offset	Name	Value / Description
0x08	bCmdId	0x03 (FLASH_ERASE)
0x09	bCmdSize	0x08
0x0c	dTransferLength	0x00000000
0x10	dAddr	The address in flash to erase, starting at this location. This must be sector (4kB) aligned
0x14	dSize	The number of bytes to erase. This must be an exact multiple of sectors (4kB)

2.8.5.4.4. READ (0x84)

Read a contiguous memory (Flash or RAM or ROM) range from the RP2040

Table 178. PICOBEST
Read memory command (Flash, RAM, ROM) structure

Offset	Name	Value / Description
0x08	bCmdId	0x84 (READ)
0x09	bCmdSize	0x08
0x0c	dTransferLength	Must be the same as dSize
0x10	dAddr	The address to read from. May be in Flash or RAM or ROM
0x14	dSize	The number of bytes to read

2.8.5.4.5. WRITE (0x05)

Writes a contiguous memory range of memory (Flash or RAM) on the RP2040.

Table 179. PICOBEST
Write memory command (Flash, RAM) structure

Offset	Name	Value / Description
0x08	bCmdId	0x05 (WRITE)
0x09	bCmdSize	0x08
0x0c	dTransferLength	Must be the same as dSize

0x0c	dTransferLength	0x00000000	
0x10	dPC	开始执行的地址。有效值包括：	
		0x00000000 通过标准Flash启动机制进行重启	
		RAM地址 通过看门狗重启，并从RAM中指定地址开始执行	
0x14	dSP	重启后的初始堆栈指针（仅在引导进入RAM时使用）	
0x18	dDelayMS	重启前的延迟时间（毫秒）	

2.8.5.4.3. FLASH_ERASE (0x03)

擦除一段连续的Flash扇区。

表177. PICOBOOT
Flash擦除命令
结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x03 (FLASH_ERASE)
0x09	bCmdSize	0x08
0x0c	dTransferLength	0x00000000
0x10	dAddr	要擦除的Flash地址，从该地址起始。此值必须以扇区（4kB）为对齐单位。
0x14	dSize	要擦除的字节数。此值必须为扇区（4kB）的整数倍。

2.8.5.4.4. 读取 (0x84)

从RP2040读取连续的内存（Flash、RAM或ROM）范围。

表178. PICOBOOT
读取内存
命令（Flash、
RAM、ROM）结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x84 (读取)
0x09	bCmdSize	0x08
0x0c	dTransferLength	必须与dSize一致。
0x10	dAddr	读取的起始地址。地址可以位于Flash、RAM或ROM中。
0x14	dSize	要读取的字节数。

2.8.5.4.5. 写入 (0x05)

在RP2040上写入连续的内存范围（Flash或RAM）。

表179。PICOBOOT
写入内存
命令（Flash、
RAM）结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x05 (写入)
0x09	bCmdSize	0x08
0x0c	dTransferLength	必须与dSize一致。

Offset	Name	Value / Description
0x10	dAddr	The address to write from. May be in Flash or RAM, however must be page (256 byte) aligned if in Flash. Note the flash must be erased first or the results are undefined.
0x14	dSize	The number of bytes to write. If writing to flash and the size is not an exact multiple of pages (256 bytes) then the last page is zero-filled to the end.

2.8.5.4.6. EXIT_XIP (0x06)

Exit Flash XIP mode. This first initialises the SSI for serial transfers, and then issues the XIP exit sequence given in [Section 2.8.1.2](#), to attempt to make the flash responsive to standard serial SPI commands. The SSI is configured with a fixed clock divisor of /6, so the USB bootloader will drive SCLK at 8MHz.

*Table 180. PICOBOOT
Exit Execute in place
(XIP) command
structure*

Offset	Name	Value / Description
0x08	bCmdId	0x06 (EXIT_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.7. ENTER_XIP (0x07)

Enter Flash XIP mode. This configures the SSI to issue a standard `03h` serial read command, with 24 address clocks and 32 data clocks, for every XIP access. This is a slow but very widely supported way to read flash. The intent of this function is to make flash easily accessible (i.e. just access addresses in the `0x10.....` segment) without having to know the details of exactly what kind of flash is connected. This mode is suitable for executing code from flash, but is much slower than e.g. QSPI XIP access.

*Table 181. PICOBOOT
Enter Execute in place
(XIP) command*

Offset	Name	Value / Description
0x08	bCmdId	0x07 (ENTER_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.8. EXEC (0x08)

Executes a function on the device. This function takes no arguments and returns no results, so it must communicate via RAM. Execution of this method will block any other commands as well as Mass Storage Interface UF2 writes, so should only be used in exclusive mode and with extreme care (and it should save and restore registers as per the ARM EABI). This method is called from a regular (non-IRQ) context, and has a very limited stack, so the function should use its own.

*Table 182. PICOBOOT
Execute function on
device command
structure*

Offset	Name	Value / Description
0x08	bCmdId	0x08 (EXEC)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	Function address to execute at (a thumb bit will be added for you since you will have forgotten).

偏移量	名称	数值 / 描述
0x10	dAddr	写入地址。写入地址可位于Flash或RAM中，但若位于Flash，则须按页（256字节）对齐。请注意，Flash必须先行擦除，否则结果无法确定。
0x14	dSize	写入的字节数。若写入Flash且大小非页（256字节）整数倍，则最后一页将以零填充至页尾。

2.8.5.4.6. EXIT_XIP (0x06)

退出Flash XIP模式。首先初始化SSI以支持串行传输，然后依据第2.8.1.2节，发出XIP退出序列，尝试使Flash响应标准串行SPI命令。SSI配置为固定的时钟分频器/6，因而USB引导程序将以8MHz频率驱动SCLK。

表180。PICOBOOT
退出执行就地
(XIP) 命令
结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x06 (EXIT_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.7. ENTER_XIP (0x07)

进入闪存XIP模式。该配置使SSI发出标准的03h串行读取命令，每次XIP访问包含24个地址时钟和32个数据时钟。这是一种速度较慢但被广泛支持的闪存读取方式。此功能旨在使闪存易于访问（即仅需访问0x10……段地址），无需了解连接的闪存具体类型。该模式适合从闪存执行代码，但速度远低于例如QSPI XIP访问。

表181。PICOBOOT
进入执行就地
(XIP)
命令

偏移量	名称	数值 / 描述
0x08	bCmdId	0x07 (ENTER_XIP)
0x09	bCmdSize	0x00
0x0c	dTransferLength	0x00000000

2.8.5.4.8. EXEC (0x08)

在设备上执行函数。该函数无参数且无返回值，必须通过RAM进行通信。执行此方法将阻塞其他命令及大容量存储接口U F2的写入操作，故应仅在独占模式下谨慎使用（且需按ARM EABI规范保存和恢复寄存器）。

此方法在常规（非IRQ）上下文中调用，且堆栈非常有限，因此函数应使用其自身堆栈。

表182。PICOBOOT
设备命令结构中
的执行函数

偏移量	名称	数值 / 描述
0x08	bCmdId	0x08 (EXEC)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	要执行的函数地址（系统会自动为您添加thumb位，以防您遗忘）。

2.8.5.4.9. VECTORIZE_FLASH (0x09)

Requests that the vector table of flash access functions used internally by the Mass Storage and PICOBOOT interfaces be copied into RAM, such that the method implementations can be replaced with custom versions (For example, if the board uses flash that does not support standard commands)

Table 183. PICOBOOT Vectorise flash command structure

Offset	Name	Value / Description
0x08	bCmdId	0x09 (VECTORIZE_FLASH)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	Pointer to where to place vector table in RAM

Flash function vector table

```
struct {
    uint32_t size; // 28
    uint32_t (*do_flash_enter_cmd_xip)();
    uint32_t (*do_flash_exit_xip)();
    uint32_t (*do_flash_erase_sector)();
    uint32_t (*do_flash_erase_range)(uint32_t addr, uint32_t size);
    uint32_t (*do_flash_page_program)(uint32_t addr, uint8_t *data);
    uint32_t (*do_flash_page_read)(uint32_t addr, uint8_t *data);
};
```

These methods have the same signature and arguments as the corresponding flash access functions in the bootrom (see [Section 2.8.3.1.3](#)).

Note that the host must subsequently update the RAM copy of this table via an **EXEC** command running on the RP2040 as any write to RAM from the host via a PICOBOOT **WRITE** that overlaps this (now active in RAM) vector table will cause a reset to the use of the default ROM Flash function vector table.

2.8.5.5. Control Requests

The following requests are sent to the interface via the default control pipe.

2.8.5.5.1. INTERFACE_RESET (0x41)

The host sends this control request to reset the PICOBOOT interface. This command:

- Clears the HALT condition (if set) on each of the bulk endpoints
- Aborts any in-process PICOBOOT or Mass Storage transfer and any flash write (this method is the only way to kill a stuck flash transfer).
- Clears the previous command result
- Removes EXCLUSIVE_ACCESS and remounts the Mass Storage drive if it was ejected due to exclusivity.

Table 184. PICOBOOT Reset PICOBOOT interface control

bmRequestType	bRequest	wValue	wIndex	wLength	Data
01000001b	01000001b	0000h	Interface	0000h	none

This command responds with an empty packet on success.

2.8.5.4.9. VECTORIZE_FLASH (0x09)

请求将Mass Storage和PICOBOOT接口内部使用的闪存访问函数向量表复制至RAM，从而允许用自定义版本替换该方法的实现（例如，若开发板所用闪存不支持标准命令）。

表183。PICOBOOT
向量化闪存
命令结构

偏移量	名称	数值 / 描述
0x08	bCmdId	0x09 (VECTORIZE_FLASH)
0x09	bCmdSize	0x04
0x0c	dTransferLength	0x00000000
0x10	dAddr	指向RAM中放置向量表位置的指针

闪存功能向量表

```
结构体 {
    uint32_t size; // 28
    uint32_t (*do_flash_enter_cmd_xip)();
    uint32_t (*do_flash_exit_xip)();
    uint32_t (*do_flash_erase_sector)();
    uint32_t (*do_flash_erase_range)(uint32_t addr, uint32_t size);
    uint32_t (*do_flash_page_program)(uint32_t addr, uint8_t *data);
    uint32_t (*do_flash_page_read)(uint32_t addr, uint8_t *data);
};
```

这些方法的签名和参数与bootrom中对应的闪存访问函数相同（见第2.8.3.1.3节）。

请注意，主机必须随后通过在RP2040上执行的 **EXEC**命令更新该表在RAM中的副本，否则主机通过PICOBOOT **WRITE**对该（现已激活的RAM中）向量表有重叠的任何RAM写入操作，都会导致复位并恢复使用默认的ROM闪存函数向量表。

2.8.5.5 控制请求

以下请求通过默认控制管道发送至该接口。

2.8.5.5.1. INTERFACE_RESET (0x41)

主机发送此控制请求以复位 PICOBOOT 接口。该命令执行以下操作：

- 清除所有大包端点上的 HALT 状态（如果已设置）
- 中止任何正在进行的 PICOBOOT 或大容量存储传输及任何闪存写入（此方法为终止卡死闪存传输的唯一途径）。
- 清除之前的命令结果
- 解除 EXCLUSIVE_ACCESS，并在因排他访问而弹出的情况下重新挂载大容量存储驱动器。

表 184. PICOBOOT
复位 PICOBOOT
接口控制

bmRequestType	bRequest	wValue	wIndex	wLength	数据
01000001b	01000001b	0000h	接口	0000h	无

该命令成功时返回空数据包作为响应。

2.8.5.5.2. GET_COMMAND_STATUS (0x42)

Retrieve the status of the last command (which may be a command still in progress). Successful completion of a PICOBLOCK Protocol Command is acknowledged over the bulk pipe, however if the operation is still in progress or has failed (stalling the bulk pipe), then this method can be used to determine the operation's status.

Table 185. PICOBLOCK
Get last command
status control

bmRequestType	bRequest	wValue	wIndex	wLength	Data
11000001b	01000010b	0000h	Interface	0000h	none

The command responds with the following 16 byte response

Table 186. PICOBLOCK
Get last command
status control
response

Offset	Name	Description		
0x00	dToken	The user token specified with the command		
0x04	dStatusCode	OK (0)	The command completed successfully (or is still in progress)	
		UNKNOWN_CMD (1)	The ID of the command was not recognized	
		INVALID_CMD_LENGTH (2)	The length of the command request was incorrect	
		INVALID_TRANSFER_LENGTH (3)	The data transfer length was incorrect given the command	
		INVALID_ADDRESS (4)	The address specified was invalid for the command type; i.e. did not match the type Flash/RAM that the command was expecting	
		BAD_ALIGNMENT (5)	The address specified was not correctly aligned according to the requirements of the command	
		INTERLEAVED_WRITE (6)	A Mass Storage Interface UF2 write has interfered with the current operation. The command was abandoned with unknown status. Note this will not happen if you have exclusive access.	
		REBOOTING (7)	The device is in the process of rebooting, so the command has been ignored.	
		UNKNOWN_ERROR (8)	Some other error occurred.	
0x08	bCmdId	The ID of the command		
0x09	bInProgress	1 if the command is still in progress	0 otherwise	
0x0a	reserved	(6 zero bytes)		

2.9. Power Supplies

RP2040 requires five separate power supplies. However, in most applications, several of these can be combined and connected to a single power source. In a typical application, only a single 3.3V supply will be required. See [Section 2.9.7.1, "Single 3.3V Supply"](#).

The power supplies and a number of potential power supply schemes are described in the following sections. Detailed power supply parameters are provided in [Section 5.6, "Power Supplies"](#).

2.8.5.5.2. GET_COMMAND_STATUS (0x42)

检索上一个命令（可能仍在处理中）的状态。PICOBOOT协议命令的成功完成通过批量传输管道确认，若操作仍在进行或失败（导致批量管道阻塞），则可通过此方法查询操作状态。

表185。PICOBOOT
获取上一个命令
状态控制

bmRequestType	bRequest	wValue	wIndex	wLength	数据
11000001b	01000010b	0000h	接口	0000h	无

该命令返回如下16字节响应数据

表186。PICOBOOT
获取上一个命令
状态控制
响应

偏移量	名称	描述	
0x00	dToken	命令中指定的用户令牌	
0x04	dStatusCode	OK (0)	命令已成功完成（或仍在执行中）
		UNKNOWN_CMD (1)	命令ID未被识别
		INVALID_CMD_LENGTH (2)	命令请求长度不正确
		INVALID_TRANSFER_LENGTH (3)	数据传输长度与命令不符
		INVALID_ADDRESS (4)	指定地址对该命令类型无效； 即地址类型与命令预期的Flash/RAM类型不符
		BAD_ALIGNMENT (5)	指定地址未根据命令要求正确对齐
		INTERLEAVED_WRITE (6)	大容量存储接口UF2写入操作干扰了当前操作。命令因状态未知而被放弃。注意：若您拥有排他访问权限，则不会发生此情况。
		REBOOTING (7)	设备正在重启，命令已被忽略。
0x08	bCmdId	命令的标识符	
	bInProgress	命令仍在执行时，值为1	否则为0
	保留	(6个字节的零值)	

2.9. 电源供应

RP2040 需要五个独立的电源。然而，在大多数应用中，其中几个电源可以合并并连接至单一电源。在典型应用中，通常只需单一3.3V电源。详见第2.9.7.1节，“单一3.3V电源”。

以下章节将介绍电源及多种潜在的电源方案。第5.6节，“电源”，包含详细的电源参数。

2.9.1. Digital IO Supply (IOVDD)

IOVDD supplies the chip's digital IO, and should be powered at a nominal voltage between 1.8V and 3.3V. The supply voltage sets the external signal level for the digital IO and should be chosen based on the signal level required. See [Section 5.5.3, "Pin Specifications"](#) for details. All digital IOs share the same power supply and operate at the same signal level.

IOVDD should be decoupled with a 100nF capacitor close to each of the chip's IOVDD pins.

⚠️ CAUTION

If the digital IO is powered at a nominal 1.8V, the IO input thresholds should be adjusted via the [VOLTAGE_SELECT](#) register. By default, the IO input thresholds are valid when the digital IO is powered at a nominal voltage between 2.5V and 3.3V. See [Section 2.19, "GPIO"](#) for details. Powering the IO at 1.8V with input thresholds set for a 2.5V to 3.3V supply is a safe operating mode, but will result in input thresholds that do not meet specification. Powering the IO at voltages greater than a nominal 1.8V with input thresholds set for a 1.8V supply may result in damage to the chip.

2.9.2. Digital Core Supply (DVDD)

DVDD supplies the chip's core digital logic, and should be powered at a nominal 1.1V. A dedicated on-chip voltage regulator is provided to allow DVDD to be generated from the digital IO supply (IOVDD) or another nominally 1.8V to 3.3V supply. The connection between the output pin of the on-chip regulator (VREG_VOUT) and the DVDD supply pins is made off-chip, allowing DVDD to be powered from an off-chip power source if required.

DVDD should be decoupled with a 100nF capacitor close to each of the chip's DVDD pins.

2.9.3. On-Chip Voltage Regulator Input Supply (VREG_VIN)

VREG_VIN is the input supply for the on-chip voltage regulator. It should be powered at a nominal voltage between 1.8V and 3.3V. To reduce the number of external power supplies, VREG_VIN can use the same power source as the digital IO supply (IOVDD).

A 1 μ F capacitor should be connected between VREG_VIN and ground close to the chip's VREG_VIN pin.

⚠️ CAUTION

VREG_VIN also powers the chip's power-on reset and brown-out detection blocks, so it must be powered even if the on-chip voltage regulator is not used.

For more details on the on-chip voltage regulator see [Section 2.10, "Core Supply Regulator"](#).

2.9.4. USB PHY Supply (USB_VDD)

USB_VDD supplies the chip's USB PHY, and should be powered at a nominal 3.3V. To reduce the number of external power supplies, USB_VDD can use the same power source as the digital IO supply (IOVDD), assuming IOVDD is also powered at 3.3V. If IOVDD is not powered at 3.3V, a separate 3.3V supply will be required for the USB PHY, see [Section 2.9.7.3, "1.8V Digital IO with Functional USB and ADC"](#). In applications where the USB PHY is never used, USB_VDD can be tied to any supply with a nominal voltage between 1.8V and 3.3V. See [Section 2.9.7.4, "Single 1.8V Supply"](#) for an example. USB_VDD should not be left unconnected.

USB_VDD should be decoupled with a 100nF capacitor close to the chip's USB_VDD pin.

2.9.1. 数字IO供电 (IOVDD)

IOVDD 为芯片的数字IO供电，电压应为1.8V至3.3V的额定值。供电电压决定数字IO的外部信号电平，并应根据所需信号电平选择。详情请参阅第5.5.3节，“引脚规格”。所有数字输入输出(IO)共用相同电源，并在相同信号电平下运行。

应在每个芯片IOVDD引脚附近，通过100nF电容对IOVDD进行去耦。

⚠ 注意

如果数字IO以标称1.8V供电，应通过VOLTAGE_SELECT寄存器调整IO输入阈值。默认情况下，当数字IO供电电压处于2.5V至3.3V标称范围内时，IO输入阈值有效。详情见第2.19节“GPIO”。以1.8V电压供电且输入阈值设置为适用于2.5V至3.3V电源的模式为安全操作模式，但输入阈值不符合规范要求。以高于标称1.8V电压供电且输入阈值设置为1.8V电源，可能导致芯片损坏。

2.9.2. 数字核心供电 (DVDD)

DVDD为芯片核心数字逻辑供电，额定电压应为1.1V。芯片内置专用稳压器，允许DVDD由数字IO电源 (IOVDD) 或其他1.8V至3.3V额定电压的电源产生。芯片内稳压器输出引脚 (VREG_VOUT) 与DVDD供电引脚之间的连接位于芯片外部，因此DVDD在需要时可由芯片外部电源供电。

芯片每个DVDD引脚附近应并联一个100nF去耦电容。

2.9.3. 片上电压调节器输入电源 (VREG_VIN)

VREG_VIN为芯片内稳压器的输入电源。其额定输入电压应为1.8V至3.3V。为减少外部电源数量，VREG_VIN可与数字IO电源 (IOVDD) 共用。

应在靠近芯片 VREG_VIN 引脚处，于 VREG_VIN 与地之间连接一个 1 μ F 电容。

⚠ 注意

VREG_VIN 还为芯片的上电复位和欠压检测模块供电，因此即使未使用片内电压调节器，该引脚仍须供电。

有关片内电压调节器的详细信息，请参见第 2.10 节，“核心电源调节器”。

2.9.4. USB PHY供电 (USB_VDD)

USB_VDD 为芯片的 USB PHY 供电，应以标称 3.3V 电压供电。为减少外部电源数量，USB_VDD 可与数字 IO 电源 (IOVDD) 共用同一电源，前提是 IOVDD 也以 3.3V 供电。若 IOVDD 不是 3.3V 供电，则 USB PHY 需单独使用 3.3V 电源，详见第 2.9.7.3 节，“带有功能 USB 和 ADC 的 1.8V 数字 IO”。在 USB PHY 从不使用的应用中，USB_VDD 可连接至标称电压介于 1.8V 至 3.3V 之间的任一电源。示例请参见第 2.9.7.4 节，“单一 1.8V 电源”。USB_VDD 不应悬空未连接。

USB_VDD 应通过靠近芯片 USB_VDD 引脚的 100nF 电容进行去耦。

2.9.5. ADC Supply (ADC_AVDD)

ADC_AVDD supplies the chip's Analogue to Digital Converter (ADC). It can be powered at a nominal voltage between 1.8V and 3.3V, but the performance of the ADC will be compromised at voltages below 2.97V. To reduce the number of external power supplies, ADC_AVDD can use from the same power source as the digital IO supply (IOVDD).

NOTE

It is safe to supply ADC_AVDD at a higher or lower voltage than IOVDD, e.g. to power the ADC at 3.3V, for optimum performance, while supporting 1.8V signal levels on the digital IO. But the voltage on the ADC analogue inputs must not exceed IOVDD, e.g. if IOVDD is powered at 1.8V, the voltage on the ADC inputs should be limited to 1.8V. Voltages greater than IOVDD will result in leakage currents through the ESD protection diodes. See [Section 5.5.3, "Pin Specifications"](#) for details.

ADC_AVDD should be decoupled with a 100nF capacitor close to the chip's ADC_AVDD pin.

2.9.6. Power Supply Sequencing

RP2040's power supplies may be powered up or down in any order. However, small transient currents may flow in the ADC supply (ADC_AVDD) if it is powered up before, or powered down after, the digital core supply (DVDD). This will not damage the chip, but can be avoided by powering up DVDD before or at the same time as ADC_AVDD, and powering down DVDD after or at the same time as ADC_AVDD. In the most common power supply scheme, where the chip is powered from a single 3.3V supply, DVDD will be powered up shortly after ADC_AVDD due to the startup time of the on-chip voltage regulator. This is acceptable behaviour. See [Section 2.9.7.1, "Single 3.3V Supply"](#).

2.9.7. Power Supply Schemes

2.9.7.1. Single 3.3V Supply

In most applications, RP2040 will be powered from a single 3.3V supply, as shown in [Figure 16](#). The digital IO (IOVDD), USB PHY (USB_VDD) and ADC (ADC_AVDD) will be powered directly from the 3.3V supply, and the 1.1V digital core supply (DVDD) will be regulated from the 3.3V supply by the on-chip voltage regulator. Note that the regulator output pin (VREG_VOUT) must be connected to the chip's DVDD pins off-chip.

For more details on the on-chip voltage regulator see [Section 2.10, "Core Supply Regulator"](#).

2.9.5. ADC供电 (ADC_AVDD)

ADC_AVDD 为芯片的模拟到数字转换器 (ADC) 供电。其标称供电电压范围为 1.8V 至 3.3V，但低于 2.97V 时，ADC 性能将受到影响。为减少外部电源数量，ADC_AVDD 可使用与数字 IO 供电 (IOVDD) 相同的电源。

① 注意

ADC_AVDD 供电电压可高于或低于 IOVDD，例如，可将 ADC 供电至 3.3V 以实现最佳性能，同时支持数字 IO 的 1.8V 信号电平。但 ADC 模拟输入端电压不得超过 IOVDD。例如，若 IOVDD 为 1.8V，则 ADC 输入端电压应限制在 1.8V。

高于 IOVDD 的电压将导致通过 ESD 保护二极管的泄漏电流。详见第 5.5.3 节，“引脚规格”。

应在芯片 ADC_AVDD 引脚附近通过一个 100nF 电容对 ADC_AVDD 进行去耦。

2.9.6. 电源顺序

RP2040 的电源可按任意顺序上电或断电。然而，若 ADC 电源 (ADC_AVDD) 在数字核心电源 (DVDD) 之前上电，或在数字核心电源断电后断电，ADC 电源可能出现短暂瞬态电流。此情况不会损害芯片，但可通过使 DVDD 在 ADC_AVDD 之前或同时上电，以及在 ADC_AVDD 之后或同时断电来避免。在最常见的电源方案中，芯片由单一 3.3V 电源供电时，因片内电压调节器的启动时间，DVDD 会在 ADC_AVDD 之后短暂延迟上电。此为可接受的行为。详见第 2.9.7.1 节，“单一 3.3V 电源”。

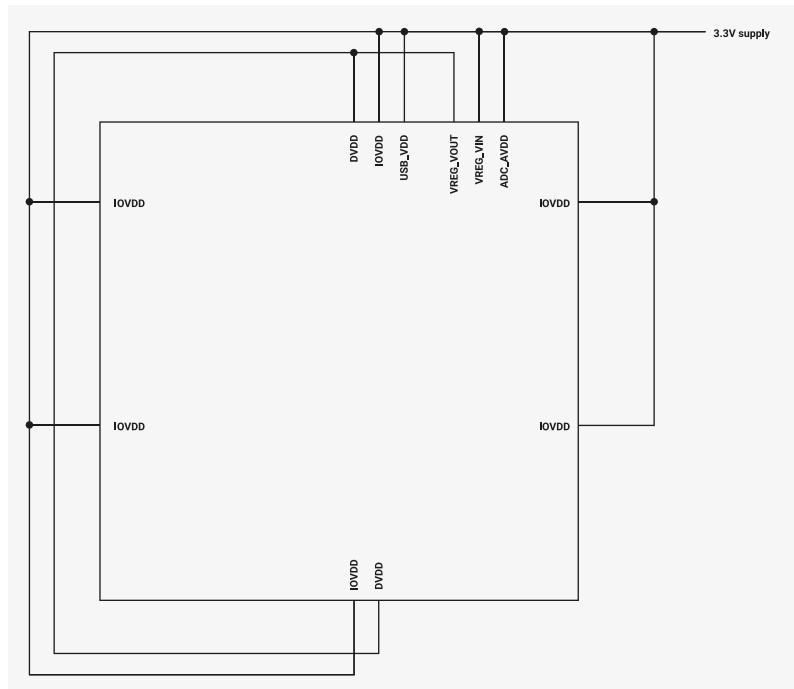
2.9.7. 电源方案

2.9.7.1. 单一 3.3V 电源

在大多数应用中，RP2040 将由单一 3.3V 电源供电，如图 16 所示。数字 IO (IOVDD)、USB PHY (USB_VDD) 及 ADC (ADC_AVDD) 将直接由 3.3V 电源供电，1.1V 数字核心电源 (DVDD) 则由片内电压调节器从 3.3V 电源调节得到。注意，调节器输出引脚 (VREG_VOUT) 必须连接至芯片片外的 DVDD 引脚。

有关片内电压调节器的详细信息，请参见第 2.10 节，“核心电源调节器”。

Figure 16. powering the chip from a single 3.3V supply
(simplified diagram omitting decoupling components)



2.9.7.2. External Core Supply

The digital core (DVDD) can be powered directly from an external 1.1V supply, rather than from the on-chip regulator, as shown in [Figure 17](#). This approach may make sense if a suitable external regulator is available elsewhere in the system, or for low power applications where an efficient switched-mode regulator could be used instead of the less efficient linear on-chip voltage regulator.

If an external core supply is used, the output of on-chip voltage regulator (VREG_VOUT) should be left unconnected. However, power must still be provided to the regulator input (VREG_VIN) to supply the chip's power-on reset and brown-out detection blocks. The on-chip voltage regulator will power-on as soon as VREG_VIN is available, but can be shutdown under software control once the chip is out of reset. See [Section 2.10, "Core Supply Regulator"](#) for details.

Figure 17. using an external core supply

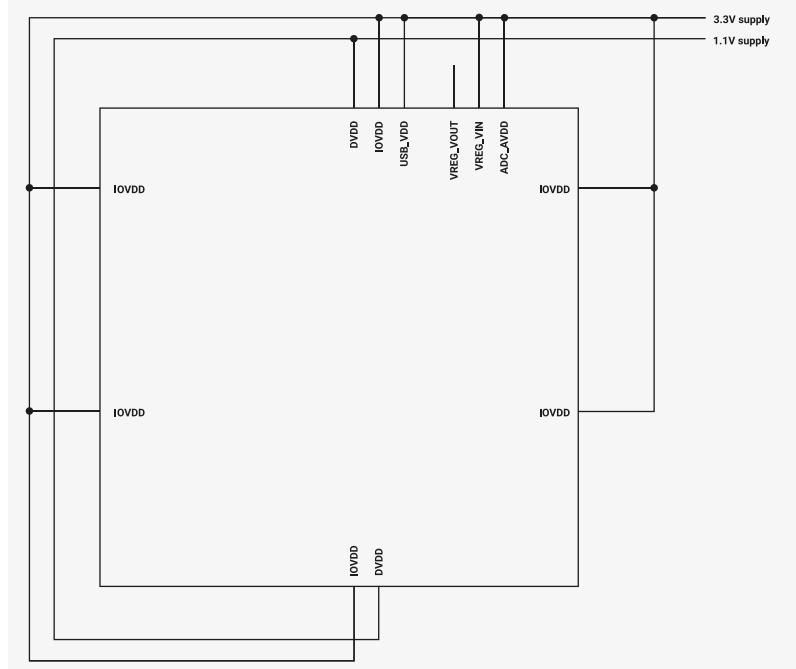
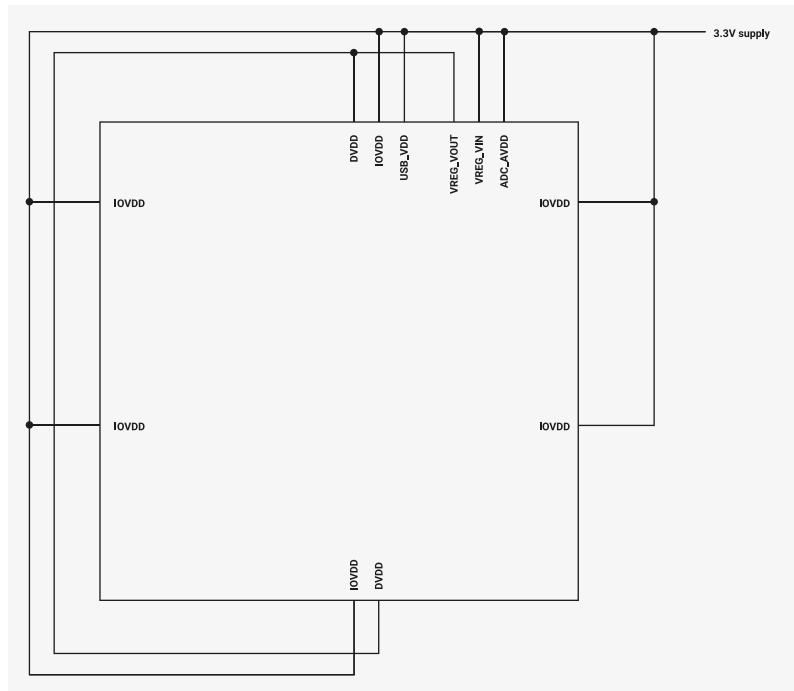


图16 采用单一3.3V电源为芯片供电
(简化示意图省略去耦元件)



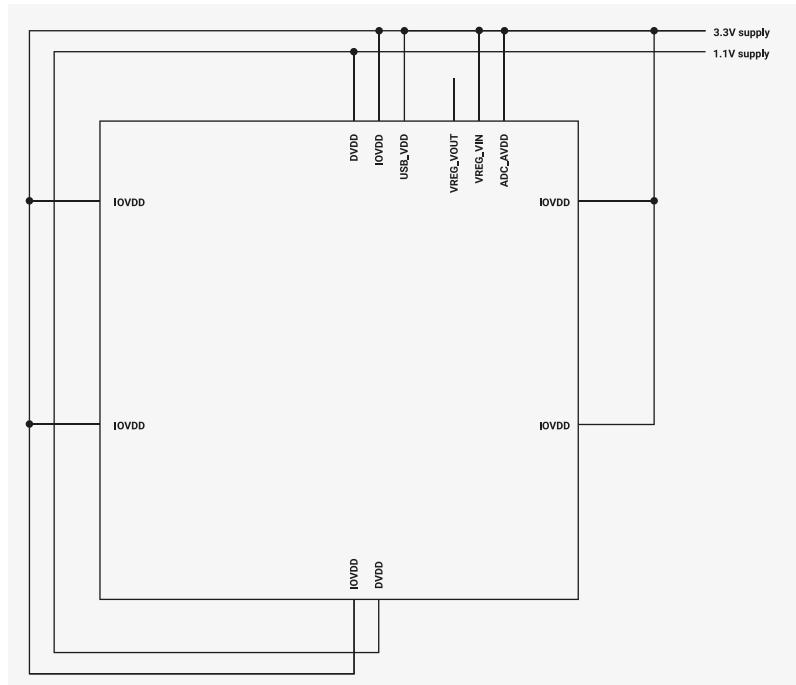
2.9.7.2. 外部核心供电

数字核心（DVDD）可直接由外部1.1V电源供电，而非来自片内调节器，如图17所示。若系统中存在合适的外部稳压器，或对于可用高效开关调节器取代低效线性片上稳压器的低功耗应用，该方案可能具有合理性。

若采用外部核心电源，则片上稳压器输出端（VREG_VOUT）应保持悬空。

但仍须向稳压器输入端（VREG_VIN）供电，以支持芯片的上电复位和欠压检测模块。片上稳压器在VREG_VIN可用时即自动上电，但芯片退出复位后可通过软件控制关闭。详见第2.10节，“核心电源稳压器”。

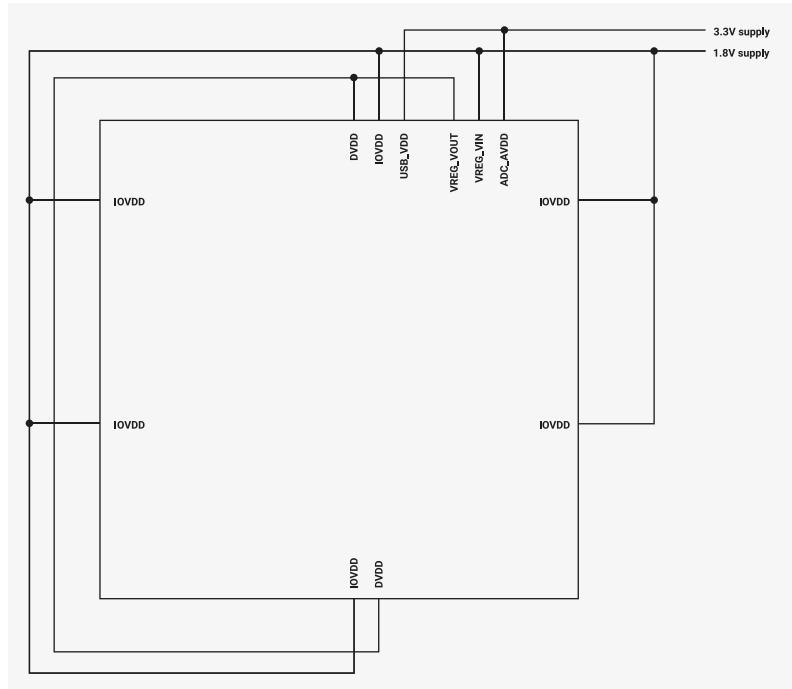
图17。使用外部核心电源



2.9.7.3. 1.8V Digital IO with Functional USB and ADC

Applications with digital IO signal levels less than 3.3V will require a separate 3.3V supply for the USB PHY and ADC, as the USB PHY does not meet specification at voltages below 3.135V and ADC performance is compromised at voltages below 2.97V. [Figure 18](#) shows an example application with the digital IO (IOVDD) powered at 1.8V and a separate 3.3V supply for the USB PHY (USB_VDD) and ADC (ADC_AVDD). In this example, the voltage regulator input (VREG_VIN) is connected to the 1.8V supply, though it could equally have been connected to the 3.3V supply. Connecting it to the 1.8V supply will reduce overall power consumption if the 1.8V supply is generated by an efficient switched-mode regulator.

*Figure 18. supporting
1.8V IO while using
USB and the ADC*



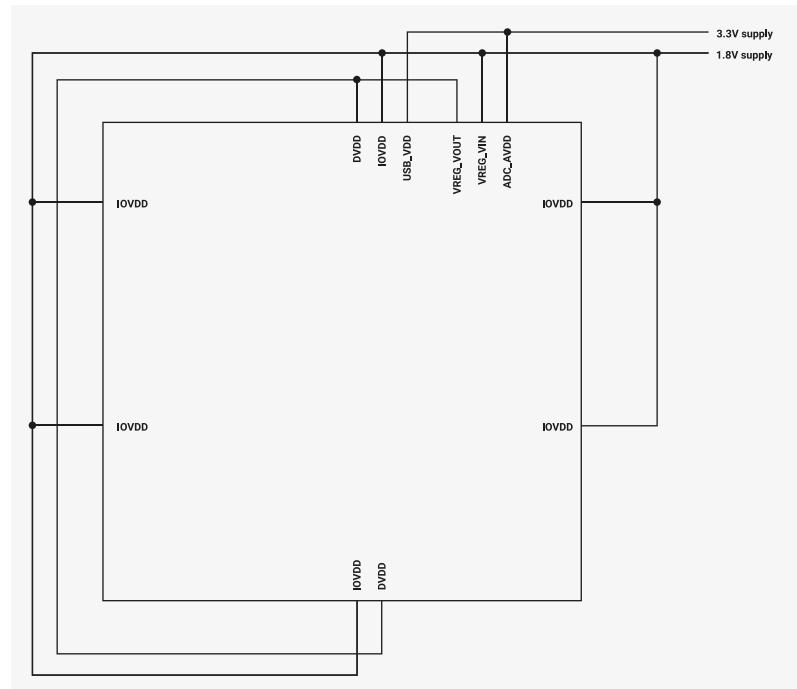
2.9.7.4. Single 1.8V Supply

If a functional USB PHY and optimum ADC performance are not required, RP2040 can be powered from a single supply of less than 3.3V. [Figure 19](#) shows an example with a single 1.8V supply. In this example, the core supply (DVDD) is regulated from the 1.8V supply by the on-chip voltage regulator.

2.9.7.3. 具功能性USB和ADC的1.8V数字IO

数字IO信号电平低于3.3V的应用需为USB PHY和ADC提供独立3.3V电源，因USB PHY在低于3.135V时不符合规格，且ADC在低于2.97V时性能受影响。图18示例中，数字IO（IOVDD）供电为1.8V，USB PHY（USB_VDD）及ADC（ADC_AVDD）采用独立3.3V电源。在本示例中，稳压器输入端（VREG_VIN）连接至1.8V电源，亦可连接至3.3V电源。若1.8V电源由高效开关稳压器产生，连接至该电源可降低整体功耗。

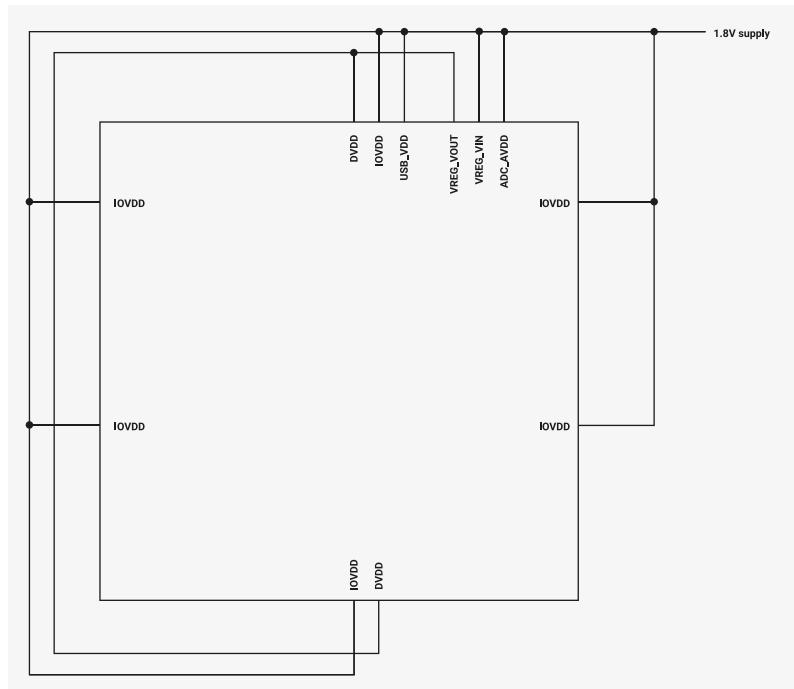
图18。在使用USB和ADC时支持1.8V IO



2.9.7.4 单一1.8V电源

若无需功能性USB PHY及最佳ADC性能，RP2040可由低于3.3V的单一电源供电。图19示例说明了单一1.8V电源的应用。在本示例中，核心电源（DVDD）由片上稳压器从1.8V电源调节获得。

Figure 19. powering
the chip from a single
1.8V supply



2.10. Core Supply Regulator

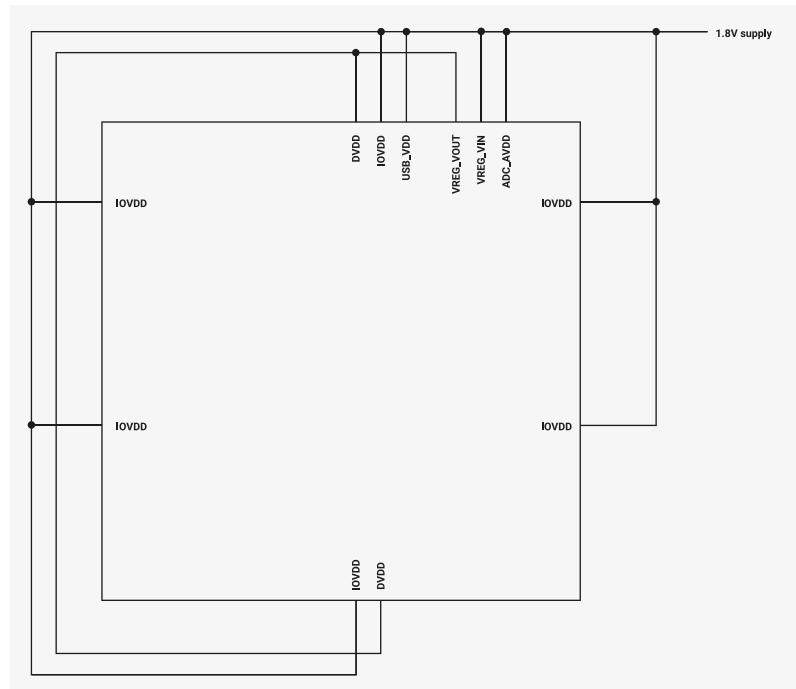
RP2040 includes an on-chip voltage regulator, allowing the digital core supply (DVDD) to be generated from an external, nominally 1.8V to 3.3V, power supply. In most cases, the regulator's input supply will share an external power source with the chip's digital IO supply IOVDD, simplifying the overall power supply requirements.

To allow the chip to start up, the voltage regulator is enabled by default and will power-on as soon as its input supply is available. Once the chip is out of reset, the regulator can be disabled, placed into a high impedance state, or have its output voltage adjusted, under software control. The output voltage can be set in the range 0.80V to 1.30V in 50mV steps, but is set to a nominal 1.1V at initial power-on, or after a reset event. The voltage regulator can supply up to 100mA.

Although intended to provide the chip's digital core supply (DVDD), the voltage regulator can be used for other purposes if DVDD is powered directly from an external power supply.

2.10.1. Application Circuit

图19。芯片由单
一1.8V电源供电



2.10. 核心供电调节器

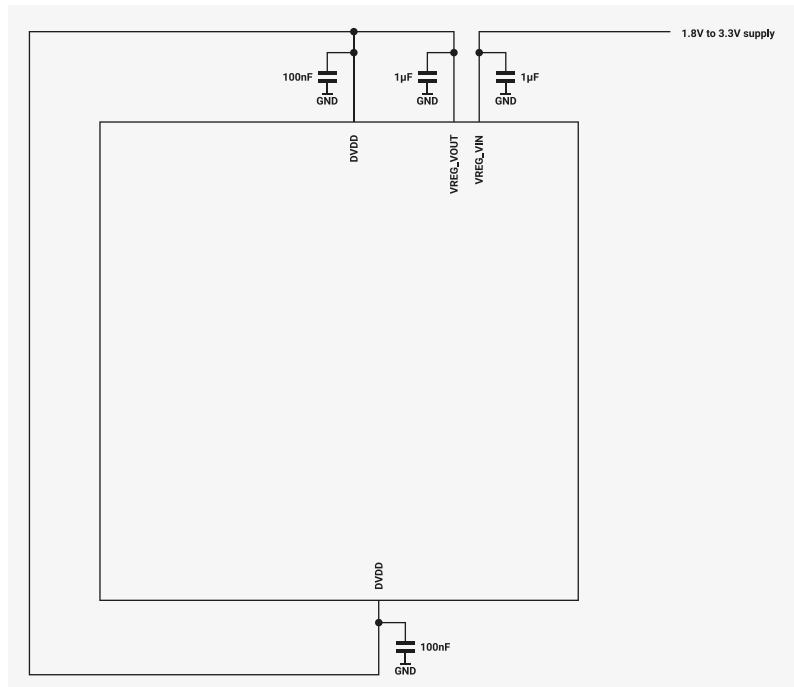
RP2040 包含片上电压调节器，允许数字核心电源（DVDD）由外部标称1.8V至3.3V电源生成。在大多数情况下，调节器的输入电源将与芯片的数字 IO 电源（IOVDD）共享外部电源，从而简化整体电源需求。

为使芯片能够启动，电压调节器默认启用，并在其输入电源可用时立即上电。芯片退出复位后，调节器可通过软件控制被禁用、置于高阻态或调整输出电压。输出电压可在0.80V至1.30V范围内以50mV步进设置，但初始上电或复位后，输出电压设定为标称1.1V。电压调节器最高可提供100mA电流。

虽然该电压调节器旨在为芯片数字核心电源（DVDD）供电，但若DVDD由外部电源直接供电，该调节器亦可用于其他用途。

2.10.1. 应用电路

Figure 20. voltage regulator application circuit



The regulator must have $1\mu\text{F}$ capacitors placed close to its input (VREG_VIN) and output (VREG_VOUT) pins.

2.10.2. Operating Modes

The voltage regulator operates in one of three modes. The mode to be used being selected by writing to the EN and HIZ fields in the VREG register, as shown in [Table 187](#). At initial power-on, or following a reset event, the voltage regulator will be in *Normal Operation* mode.

Table 187. Voltage Regulator Mode Select

Mode	EN	HIZ
Normal Operation ^a	1	0
High Impedance	1	1
Shutdown	0	X

^a the voltage regulator will be in normal mode at initial power-on or following a reset event

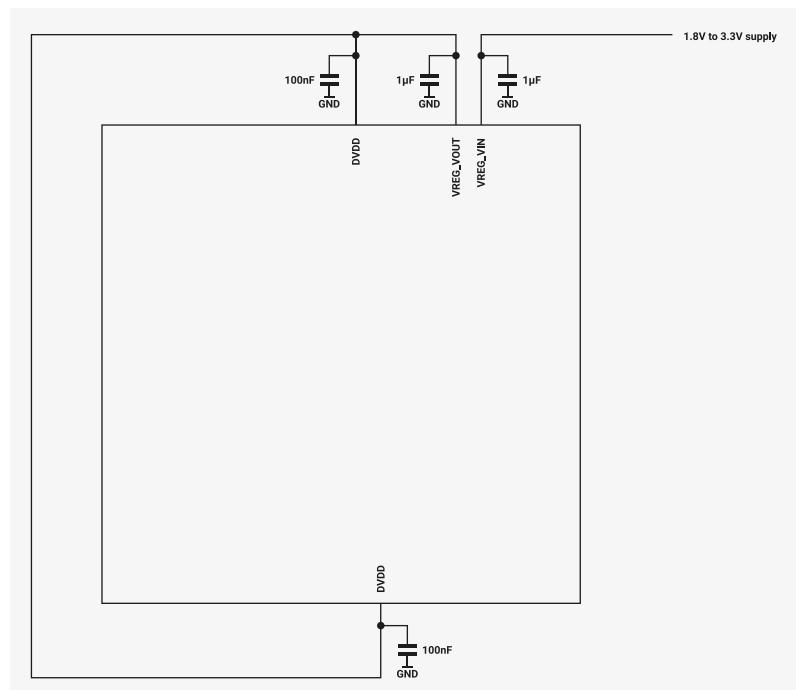
2.10.2.1. Normal Operation Mode

In Normal Operation mode, the voltage regulator's output is in regulation at the selected voltage, and the regulator is able to supply power.

2.10.2.2. High Impedance Mode

In High Impedance mode, the voltage regulator is disabled and its output pin (VREG_VOUT) is set to a high impedance state. In this mode, the regulator's power consumption is minimised. This mode allows a load connected to VREG_VOUT to be powered from a power source other than the on-chip regulator. This could allow, for example, the load to be initially powered from the on-chip voltage regulator, and then switched to an external regulator under software control. The external regulator would also need to support a high impedance mode, with only one regulator supplying the load at a time. The supply voltage is maintained by the regulator's output capacitor during the brief period when both regulators are in high impedance mode.

图20。电压
调节器应用
电路



调节器的输入（VREG_VIN）和输出（VREG_VOUT）引脚附近必须放置 $1\mu F$ 电容。

2.10.2. 工作模式

电压调节器在三种模式之下工作。所选模式通过向VREG寄存器中的 EN和 HIZ字段写入实现，如表187所示。在初始上电或复位事件后，电压调节器处于正常工作模式。

表187。电压
调节器模式选择

模式	EN	HIZ
正常工作 ^a	1	0
高阻抗	1	1
关断	0	X

^a t 在初始上电或复位事件后，电压调节器处于正常模式

2.10.2.1. 正常工作模式

在正常运行模式下，电压调节器的输出电压保持在所选电压范围内，且调节器能够稳定供电。

2.10.2.2. 高阻抗模式

在高阻抗模式下，电压调节器被禁用，其输出引脚（VREG_VOUT）处于高阻抗状态。在该模式下，调节器的功耗降至最低。此模式允许连接到 VREG_VOUT 的负载由片内调节器以外的电源供电。例如，负载可先由片内电压调节器供电，然后通过软件控制切换至外部调节器供电。外部调节器亦须支持高阻抗模式，且任一时刻仅允许一个调节器为负载供电。当两者均处于高阻抗模式的短暂期间内，输出电容器维持调节器的供电电压。

2.10.2.3. Shutdown Mode

In Shutdown mode, the voltage regulator is disabled, power consumption is minimized and the regulator's output pin (VREG_VOUT) is pulled to 0V.

Shutdown mode is only useful if the voltage regulator is not providing the RP2040's digital core supply (DVDD). If the regulator is supplying DVDD, and brown-out detection is enabled, entering shutdown mode will cause a reset event and the voltage regulator will return to normal mode. If brown-out detection isn't enabled, the voltage regulator will shut down and will remain in shutdown mode until its input supply (VREG_VIN) is power cycled.

2.10.3. Output Voltage Select

The required output voltage can be selected by writing to the `VSEL` field in the `VREG` register. The voltage regulator's output voltage can be set in the range 0.80V to 1.30V in 50mV intervals. The regulator output voltage is set to 1.1V at initial power-on or following a reset event. For details, see the `VREG` register description.

Note that RP2040 may not operate reliably with its digital core supply (DVDD) outside of operating conditions (see [Section 5.6](#)); it is recommended to set the regulator to 1.10V for most applications.

2.10.4. Status

The `VREG` register contains a single status field, `ROK`, which indicates whether the voltage regulator's output is being correctly regulated.

At power on, `ROK` remains low until the regulator has started up and the output voltage reaches the `ROK` assertion threshold (`ROKTH,ASSERT`). It then remains high until the voltage drops below the `ROK` deassertion threshold (`ROKTH,DEASSERT`), remaining low until the output voltage is above the assertion threshold again. `ROKTH,ASSERT` is nominally 90% of the selected output voltage, 0.99V if the selected output voltage is 1.1V, and `ROKTH,DEASSERT` is nominally 87% of the selected output voltage, 0.957V if the selected output voltage is 1.1V.

Note that adjusting the output voltage to a higher voltage will cause `ROK` to go low until the assertion threshold for the higher voltage is reached. `ROK` will also go low if the regulator is placed in high impedance mode.

2.10.5. Current Limit

The voltage regulator includes a current limit to prevent the load current exceeding the maximum rated value. The output voltage will not be regulated and will drop below the selected value when the current limit is active.

2.10.6. List of Registers

The voltage regulator shares a register address space with the chip-level reset subsystem. The registers for both subsystems are listed here. Only, the `VREG` register is part of the voltage register subsystem. The `BOD` and `CHIP_RESET` registers are part of the chip-level reset subsystem. The shared address space is referred to as `vreg_and_chip_reset` elsewhere in this document.

The `VREG_AND_CHIP_RESET` registers start at a base address of `0x40064000` (defined as `VREG_AND_CHIP_RESET_BASE` in SDK).

Table 188. List of VREG_AND_CHIP_RESET registers

Offset	Name	Info
0x0	<code>VREG</code>	Voltage regulator control and status
0x4	<code>BOD</code>	brown-out detection control
0x8	<code>CHIP_RESET</code>	Chip reset control and status

2.10.2.3. 关闭模式

在关机模式下，电压调节器被禁用，功耗降至最低，调节器的输出引脚（VREG_VOUT）被拉至0V。

关机模式仅在电压调节器未为RP2040数字核心电源（DVDD）供电时才有效。如果调节器为DVDD供电且启用了欠压检测，进入关机模式将触发复位事件，电压调节器将恢复至正常模式。若未启用欠压检测，电压调节器将关闭，并保持在关机模式，直至其输入电源（VREG_VIN）断电重启。

2.10.3. 输出电压选择

所需输出电压可通过写入VREG寄存器中的 **VSEL** 字段进行选择。电压调节器的输出电压可在0.80V至1.30V范围内，以50mV为步进进行设置。调节器的输出电压在初次上电或复位后设置为1.1V。有关详细信息，请参阅VREG寄存器说明。

请注意，RP2040在其数字核心电源（DVDD）超出工作条件时可能无法稳定运行（参见第5.6节）；建议大多数应用中将稳压器设置为**1.10V**。

2.10.4. 状态

VREG寄存器包含一个状态字段，**ROK**，用以指示电压稳压器输出是否处于正确调节状态。

上电时，**ROK**保持低电平，直至稳压器启动且输出电压达到**ROK_{TH,ASSERT}**。随后，**ROK**保持高电平，直到电压降至**ROK_{TH,DEASSERT}**；之后保持低电平，直至输出电压再次超过断言阈值。**ROK_{TH,ASSERT}**的名义值为所选输出电压的90%，当输出电压为1.1V时，为0.99V；**ROK_{TH,DEASSERT}**的名义值为所选输出电压的87%，当输出电压为1.1V时，为0.957V。

请注意，将输出电压调高至更高电压时，**ROK**将保持低电平，直到达到更高电压的断言阈值。当稳压器处于高阻抗模式时，**ROK**亦将保持低电平。

2.10.5. 电流限制

该电压稳压器设有电流限制，以防止负载电流超过最大额定值。当电流限制生效时，输出电压将不受调节，且会低于所选值。

2.10.6. 寄存器列表

电压稳压器与芯片级复位子系统共享寄存器地址空间。两个子系统的寄存器列表见此处。仅VREG寄存器属于电压寄存器子系统。BOD和CHIP_RESET寄存器属于芯片级复位子系统。本文档其他部分将该共享地址空间称为**vreg_and_chip_reset**。

VREG_AND_CHIP_RESET寄存器起始地址为 **0x40064000**（在SDK中定义为VREG_AND_CHIP_RESET_BASE）。

表188。VREG_AND_CHIP_RESET寄存器列表

偏移量	名称	说明
0x0	VREG	电压调节器控制与状态
0x4	BOD	欠压检测控制
0x8	CHIP_RESET	芯片复位控制与状态

VREG_AND_CHIP_RESET: VREG Register

Offset: 0x0

Description

Voltage regulator control and status

Table 189. VREG Register

Bits	Description	Type	Reset
31:13	Reserved.	-	-
12	ROK: regulation status 0=not in regulation, 1=in regulation	RO	0x0
11:8	Reserved.	-	-
7:4	VSEL: output voltage select 0000 to 0101 - 0.80V 0110 - 0.85V 0111 - 0.90V 1000 - 0.95V 1001 - 1.00V 1010 - 1.05V 1011 - 1.10V (default) 1100 - 1.15V 1101 - 1.20V 1110 - 1.25V 1111 - 1.30V	RW	0xb
3:2	Reserved.	-	-
1	HIZ: high impedance mode select 0=not in high impedance mode, 1=in high impedance mode	RW	0x0
0	EN: enable 0=not enabled, 1=enabled	RW	0x1

VREG_AND_CHIP_RESET: BOD Register

Offset: 0x4

Description

brown-out detection control

Table 190. BOD Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-

VREG_AND_CHIP_RESET: VREG寄存器

偏移: 0x0

描述

电压调节器控制与状态

表 189. VREG 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	ROK: 调节状态 0=未调节, 1=调节中	只读	0x0
11:8	保留。	-	-
7:4	VSEL: 输出电压选择 0000 至0101 - 0.80V 0110 - 0.85V 0111 - 0.90V 1000 - 0.95V 1001 - 1.00V 1010 - 1.05V 1011 - 1.10V (默认) 1100 - 1.15V 1101 - 1.20V 1110 - 1.25V 1111 - 1.30V	读写	0xb
3:2	保留。	-	-
1	HIZ: 高阻态模式选择 0=非高阻态模式, 1=高阻态模式	读写	0x0
0	EN: 使能 0=未使能, 1=已使能	读写	0x1

VREG_AND_CHIP_RESET: BOD 寄存器

偏移: 0x4

说明

欠压检测控制

表 190. BOD 寄存器

位	描述	类型	复位值
31:8	保留。	-	-

Bits	Description	Type	Reset
7:4	VSEL : threshold select 0000 - 0.473V 0001 - 0.516V 0010 - 0.559V 0011 - 0.602V 0100 - 0.645V 0101 - 0.688V 0110 - 0.731V 0111 - 0.774V 1000 - 0.817V 1001 - 0.860V (default) 1010 - 0.903V 1011 - 0.946V 1100 - 0.989V 1101 - 1.032V 1110 - 1.075V 1111 - 1.118V	RW	0x9
3:1	Reserved.	-	-
0	EN : enable 0=not enabled, 1=enabled	RW	0x1

VREG_AND_CHIP_RESET: CHIP_RESET Register

Offset: 0x8

Description

Chip reset control and status

Table 191.
CHIP_RESET Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24	PSM_RESTART_FLAG : This is set by psm_restart from the debugger. Its purpose is to branch bootcode to a safe mode when the debugger has issued a psm_restart in order to recover from a boot lock-up. In the safe mode the debugger can repair the boot code, clear this flag then reboot the processor.	WC	0x0
23:21	Reserved.	-	-
20	HAD_PSM_RESTART : Last reset was from the debug port	RO	0x0
19:17	Reserved.	-	-
16	HAD_RUN : Last reset was from the RUN pin	RO	0x0
15:9	Reserved.	-	-
8	HAD POR : Last reset was from the power-on reset or brown-out detection blocks	RO	0x0
7:0	Reserved.	-	-

位	描述	类型	复位值
7:4	VSEL : 阈值选择 0000 - 0.473V 0001 - 0.516V 0010 - 0.559V 0011 - 0.602V 0100 - 0.645V 0101 - 0.688V 0110 - 0.731V 0111 - 0.774V 1000 - 0.817V 1001 - 0.860V (默认) 1010 - 0.903V 1011 - 0.946V 1100 - 0.989V 1101 - 1.032V 1110 - 1.075V 1111 - 1.118V	读写	0x9
3:1	保留。	-	-
0	EN : 使能 0=未使能, 1=已使能	读写	0x1

VREG_AND_CHIP_RESET: CHIP_RESET 寄存器

偏移: 0x8

描述

芯片复位控制与状态

表191。
CHIP_RESET 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	PSM_RESTART_FLAG : 此标志由调试器中的 psm_restart 设置。 其目的是在调试器执行 psm_restart 以从启动死锁中恢复时, 使启动代码跳转至安全模式。 在安全模式下, 调试器可修复启动代码, 清除此标志, 然后重新启动处理器。	WC	0x0
23:21	保留。	-	-
20	HAD_PSM_RESTART : 上次复位来源于调试端口	只读	0x0
19:17	保留。	-	-
16	HAD_RUN : 上次复位来源于RUN引脚	只读	0x0
15:9	保留。	-	-
8	HAD_POR : 上次复位来源于上电复位或欠压检测模块	只读	0x0
7:0	保留。	-	-

2.10.7. Detailed Specifications

Table 192. Voltage Regulator Detailed Specifications

Parameter	Description	Min	Typ	Max	Units
V_{VREG_VIN}	input supply voltage	1.63	1.8 - 3.3	3.63	V
ΔV_{VREG_VOUT}	output voltage variation	-3		+3	% of selected output voltage
I_{MAX}	output current			100	mA
I_{LIMIT}	current limit	150	350	450	mA
$ROK_{TH,ASSERT}$	ROK assertion threshold	87	90	93	% of selected output voltage
$ROK_{TH,DEASSERT}$	ROK deassertion threshold	84	87	90	% of selected output voltage
$t_{POWER-ON}^a$	power-up time		275	350	μs

^a values will vary with load current and capacitance on VREG_VOUT. Conditions: EN = 1, load current = 0mA, VREG_VIN ramps up in 100 μ s

2.11. Power Control

RP2040 provides a range of options for reducing dynamic power:

- Top-level clock gating of individual peripherals and functional blocks
- Automatic control of top-level clock gates based on processor sleep state
- On-the-fly changes to system clock frequency or system clock source (e.g. switch to internal ring oscillator, and disable PLLs and crystal oscillator)
- Zero-dynamic-power DORMANT state, waking on GPIO event or RTC IRQ

All digital logic on RP2040 is in a single core power domain. The following options are available for static power reduction:

- Placing memories into state-retaining power down state
- Power gating on peripherals that support this, e.g. ADC, temperature sensor

2.11.1. Top-level Clock Gates

Each clock domain (for example, the system clock) may drive a large number of distinct hardware blocks, not all of which may be required at once. To avoid unnecessary power dissipation, each individual endpoint of each clock (for example, the UART system clock input) may be disabled at any time.

Enabling and disabling a clock gate is glitch-free. If a peripheral clock is temporarily disabled, and subsequently re-enabled, the peripheral will be in the same state as prior to the clock being disabled. No reset or reinitialisation should be required.

Clock gates are controlled by two sets of registers: the WAKE_ENx registers (starting at [WAKE_EN0](#)) and SLEEP_ENx registers (starting at [SLEEP_EN0](#)). These two sets of registers are identical at the bit level, each possessing a flag to control each clock endpoint. The WAKE_EN registers specify which clocks are enabled whilst the system is awake, and the SLEEP_ENx registers select which clocks are enabled while the processor is in the SLEEP state ([Section 2.11.2](#)).

The two Cortex-M0+ processors do not have externally-controllable clock gates. Instead, the processors gate the clocks of their subsystems autonomously, based on execution of [WFI/WFE](#) instructions, and external Event and IRQ signals.

2.10.7. 详细规格

表192。电压
调节器详细
规格

参数	描述	最小值	典型值	最大值	单位
V_{VREG_VIN}	输入电源 电压	1.63	1.8 - 3.3	3.63	V
ΔV_{VREG_VOUT}	输出电压 变化	-3		+3	选定输出电压的 百分比
I_{MAX}	输出电流			100	mA
电流限制LIMIT	电流限制	150	350	450	mA
$ROK_{TH.ASSERT}$	ROK 断言 阈值	87	90	93	选定输出电压的 百分比
$ROK_{TH.DEASSERT}$	ROK 解除断言 阈值	84	87	90	选定输出电压的 百分比
$t_{\text{通电时间}}^a$	通电时间		275	350	μ秒

^a数值会因负载电流及VREG_VOUT上的电容而变化。条件：EN=1，负载电流=0mA，VREG_VIN在100μs内上升

2.11. 电源控制

RP2040提供多种降低动态功耗的选项：

- 对各单独外设及功能模块的顶层时钟门控
- 基于处理器睡眠状态的顶层时钟门控自动控制
- 系统时钟频率或时钟源的动态切换（例如切换至内部环振荡器，禁用PLL和晶振）
- 零动态功耗的休眠（DORMANT）状态，可由GPIO事件或RTC中断唤醒

RP2040上的所有数字逻辑均处于单一核心电源域。以下选项可用于降低静态功耗：

- 将存储器置于保持状态的断电模式
- 对支持此功能的外围设备执行电源门控，例如ADC和温度传感器

2.11.1. 顶层时钟门控

每个时钟域（例如系统时钟）可能驱动大量不同的硬件模块，且不一定同时全部需要。为避免不必要的功率消耗，每个时钟的单独终端（例如UART系统时钟输入）可以随时被禁用。

启用和禁用时钟门控操作无毛刺。若外围设备时钟被暂时禁用，随后重新启用，外围设备将保持禁用前的状态。无需复位或重新初始化。

时钟门控由两组寄存器控制：WAKE_ENx寄存器（自WAKE_EN0起）和SLEEP_ENx寄存器（自SLEEP_EN0起）。这两组寄存器位级完全相同，每个位均含控制相应时钟终端的标志位。WAKE_EN寄存器指定系统处于唤醒状态时启用的时钟，SLEEP_ENx寄存器则选择处理器处于SLEEP状态时启用的时钟（参见第2.11.2节）。

两个Cortex-M0+处理器不具有外部可控的时钟门控。处理器会基于执行的WFI/WFE指令以及外部事件（Event）和中断请求（IRQ）信号，自主控制其子系统的时钟门控。

2.11.2. SLEEP State

RP2040 enters the SLEEP state when all of the following are true:

- Both processors are asleep (e.g. in a [WFE](#) or [WFI](#) instruction)
- The system DMA has no outstanding transfers on any channel

RP2040 exits the SLEEP state when either processor is awoken by an interrupt.

When in the SLEEP state, the top-level clock gates are masked by the SLEEP_ENx registers (starting at [SLEEP_EN0](#)), rather than the WAKE_ENx registers. This permits more aggressive pruning of the clock tree when the processors are asleep.

NOTE

Though it is possible for a clock to be enabled during SLEEP and disabled outside of SLEEP, this is generally not useful

For example, if the system is sleeping until a character interrupt from a UART, the entire system except for the UART can be clock-gated (SLEEP_ENx = all-zeroes except for CLK_SYS_UART0 and CLK_PERI_UART0). This includes system infrastructure such as the bus fabric.

When the UART asserts its interrupt, and wakes a processor, RP2040 leaves SLEEP mode, and switches back to the WAKE_ENx clock mask. At the minimum this should include the bus fabric, and the memory devices containing the processor's stack and interrupt vectors.

A system-level clock request handshake holds the processors off the bus until the clocks are re-enabled.

2.11.3. DORMANT State

The DORMANT state is a true zero-dynamic-power sleep state, where all clocks (and all oscillators) are disabled. The system can awake from the DORMANT state upon a GPIO event (high/low level or rising/falling edge), or an RTC interrupt: this restarts one of the oscillators (either ring oscillator or crystal oscillator), and ungates the oscillator output once it is stable. System state is retained, so code execution resumes immediately upon leaving the DORMANT state.

Note that, if relying on the RTC ([Section 4.8](#)) to wake from the DORMANT state, the RTC must have some external clock source. The RTC accepts clock frequencies as low as 1Hz.

Note also that DORMANT does not halt PLLs. To avoid unnecessary power dissipation, software should power down PLLs before entering the DORMANT state, and power up and reconfigure the PLLs again after exiting.

The DORMANT state is entered by writing a keyword to the DORMANT register in whichever oscillator is active: ring oscillator ([Section 2.17](#)) or crystal oscillator ([Section 2.16](#)). If both are active then the one providing the processor clock must be stopped last because it will stop software from executing.

2.11.4. Memory Power Down

The main system memories (SRAM0...5, mapped to bus addresses [0x20000000](#) to [0x20041fff](#)), as well as the USB DPRAM, can be powered down via the [MEMPOWERDOWN](#) register in the Syscfg registers (see [Section 2.21](#)). When powered down, memories retain their current contents, but cannot be accessed. Static power is reduced.

2.11.2. SLEEP 状态

当满足以下所有条件时，RP2040进入SLEEP状态：

- 两个处理器均处于休眠状态（例如执行 `WFE` 或 `WFI` 指令）
- 系统DMA在任一通道上均无未完成的传输

当任一处理器被中断唤醒时，RP2040即退出SLEEP状态。

处于SLEEP状态时，顶级时钟门控由SLEEP_ENx寄存器（从SLEEP_EN0开始）进行屏蔽，而非WAKE_ENx寄存器。此功能允许在处理器睡眠时对时钟树进行更严格的修剪。

注意

虽然时钟可以在SLEEP期间启用而在非SLEEP期间禁用，但此做法通常无实用价值。

例如，若系统处于睡眠状态直至UART字符中断，除UART外的整个系统均可进行时钟门控（SLEEP_ENx 除CLK_SYS_UART0和CLK_PERI_UART0外全为零），其中包括总线结构等系统基础设施。

当UART触发中断并唤醒处理器时，RP2040退出SLEEP模式，并切换回WAKE_ENx时钟掩码。至少应包含总线结构及存储处理器堆栈和中断向量的存储设备。

系统级时钟请求握手机制会在时钟重新启用前，阻止处理器访问总线。

2.11.3. 休眠状态

DORMANT状态为真正的零动态功耗睡眠状态，所有时钟及振荡器均被禁用。系统可通过GPIO事件（高/低电平或上升/下降沿）或RTC中断从DORMANT状态唤醒：该操作重新启动其中一个振荡器（环形振荡器或晶体振荡器），并在振荡器输出稳定后解除门控。系统状态被保留，代码执行将在离开DORMANT状态后立即恢复。

注意，若依赖RTC（第4.8节）从DORMANT状态唤醒，RTC必须具有外部时钟源。RTC接受低至1Hz的时钟频率。

还需注意，DORMANT状态并不会停止PLL。为避免不必要的功耗，软件应在进入DORMANT状态前关闭PLL，并在退出后重新上电及重新配置PLL。

通过向当前活动振荡器的DORMANT寄存器写入关键字进入DORMANT状态：环形振荡器（第2.17节）或晶体振荡器（第2.16节）。若两者均处于活动状态，必须最后停止为处理器提供时钟的振荡器，否则将导致软件停止执行。

2.11.4. 内存断电

主系统存储器（SRAM0...5，映射至总线地址 `0x20000000` 至 `0x20041fff`），以及USB DPRAM，可通过 Syscfg 寄存器中的 MEMPOWERDOWN 寄存器断电（详见第 2.21 节）。断电时，存储器保持当前内容，但无法被访问。静态功耗得以降低。

⚠ CAUTION

Memories must not be accessed when powered down. Doing so can corrupt memory contents.

When powering a memory back up, a 20ns delay is required before accessing the memory again.

The XIP cache (see [Section 2.6.3](#)) can also be powered down, with `CTRL.POWER_DOWN`. The XIP hardware will not generate cache accesses whilst the cache is powered down. Note that this is unlikely to produce a net power savings if code continues to execute from XIP, due to the comparatively high voltages and switching capacitances of the external QSPI bus.

2.11.5. Programmer's Model

2.11.5.1. Sleep

The `hello_sleep` example, https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c, demonstrates sleep mode. The `hello_sleep` application (and underlying functions) takes the following steps:

- Run all clocks in the system from XOSC
- Configure an alarm in the RTC for 10 seconds in the future
- Set `clk_rtc` as the only clock running in sleep mode using the `SLEEP_ENx` registers (see [SLEEP_EN0](#))
- Enable deep sleep in the processor
- Call `_wfi` on processor which will put the processor into deep sleep until woken by the RTC interrupt
- The RTC interrupt clears the alarm and then calls a user supplied callback function
- The callback function ends the example application

ℹ NOTE

It is necessary to enable deep sleep on both `proc0` and `proc1` and call `_wfi`, as well as ensure the DMA is stopped to enter sleep mode.

`hello_sleep` makes use of functions in `pico_sleep` of the [Pico Extras](#). In particular, `sleep_goto_sleep_until` puts the processor to sleep until woken up by an RTC time assumed to be in the future.

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/ip2_common/pico_sleep/sleep.c Lines 159 - 183

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // We should have already called the sleep_run_from_dormant_source function
163     // This is only needed for dormancy although it saves power running from xosc while
164     // sleeping
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCK_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;
169
170     aon_timer_enable_alarm(ts, callback, false);
171
172     stdio_flush();
173
174     // Enable deep sleep at the proc
175     processor_deep_sleep();

```

⚠ 注意

断电状态下不得访问存储器。否则可能导致存储器内容损坏。

重新上电存储器后，需延迟 20ns 后方可再次访问存储器。

XIP 缓存（详见第 2.6.3 节）亦可通过 CTRL.POWER_DOWN 实现断电。缓存断电期间，XIP 硬件不会产生缓存访问请求。注意，若代码仍从 XIP 执行，鉴于外部 QSPI 总线较高的电压和开关电容，通常无法实现净功耗节约。

2.11.5. 程序员模型

2.11.5.1 睡眠

示例hello_sleep位于https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c，演示了睡眠模式。hello_sleep 应用程序（及其底层函数）执行以下步骤：

- 将系统中所有时钟设为由 XOSC 驱动
- 在 RTC 中配置一个 10 秒后的闹钟
- 使用 SLEEP_ENx 寄存器（参见 SLEEP_EN0）设置 clk_RTC 为睡眠模式下唯一运行的时钟
- 启用处理器深度睡眠
- 在处理器上调用 `_wfi` 指令，使处理器进入深度睡眠状态，直至被 RTC 中断唤醒
- RTC 中断清除闹钟后调用用户提供的回调函数
- 回调函数结束示例应用程序

 ⓘ 注意

须在 proc0 和 proc1 上均启用深度睡眠并调用 `_wfi`，且确保 DMA 停止，方可进入睡眠模式。

hello_sleep 利用 Pico Extras 中 `pico_sleep` 的函数。特别是，`sleep_goto_sleep_until` 使处理器进入睡眠状态，直至被假定在未来的 RTC 时间唤醒。

Pico 附加功能：https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第159至183行

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // 我们应已调用过 sleep_run_from_dormant_source 函数
163 // 该功能仅在休眠状态下需要，尽管它在睡眠时通过 xosc 节约了功耗
164
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCKS_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;
169
170     aon_timer_enable_alarm(ts, callback, false);
171
172     stdio_flush();
173
174     // 在处理器上启用深度睡眠
175     processor_deep_sleep();

```

```

175
176     // Go to sleep
177     __wfi();
178 }
```

2.11.5.2. Dormant

The `hello_dormant` example, https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_dormant/hello_dormant_gpio.c, demonstrates dormant mode. The example takes the following steps:

- Run all clocks in the system from XOSC
- Configure a GPIO interrupt for the "dormant_wake" hardware which can wake both the ROSC and XOSC from dormant mode
- Put the XOSC into dormant mode which stops all processor execution (and all other clocked logic on the chip) immediately
- When GPIO 10 goes high, the XOSC is started again and execution of the program continues

`hello_dormant` uses `sleep_goto_dormant_until_pin` under the hood:

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c Lines 258 - 282

```

258 void sleep_goto_dormant_until_pin(uint gpio_pin, bool edge, bool high) {
259     bool low = !high;
260     bool level = !edge;
261
262     // Configure the appropriate IRQ at IO bank 0
263     assert(gpio_pin < NUM_BANK0_GPIOS);
264
265     uint32_t event = 0;
266
267     if (level && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_LOW_BITS;
268     if (level && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_HIGH_BITS;
269     if (edge && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_HIGH_BITS;
270     if (edge && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_LOW_BITS;
271
272     gpio_init(gpio_pin);
273     gpio_set_input_enabled(gpio_pin, true);
274     gpio_set_dormant_irq_enabled(gpio_pin, event, true);
275
276     _go_dormant();
277     // Execution stops here until woken up
278
279     // Clear the irq so we can go back to dormant mode again if we want
280     gpio_acknowledge_irq(gpio_pin, event);
281     gpio_set_input_enabled(gpio_pin, false);
282 }
```

2.12. Chip-Level Reset

2.12.1. Overview

The chip-level reset subsystem resets the whole chip, placing it in a default state. This happens at initial power-on, during a power supply brown-out event or when the chip's RUN pin is taken low. The chip can also be reset via the

```

175
176      // 进入睡眠状态
177      __wfi();
178 }

```

2.11.5.2. 休眠模式

hello_dormant 示例，https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_dormant/hello_dormant_gpio.c，演示了休眠模式。该示例执行以下步骤：

- 将系统中所有时钟设为由 XOSC 驱动
- 为“dormant_wake”硬件配置 GPIO 中断，此硬件可唤醒处于休眠模式的 ROSC 和 XOSC
- 将 XOSC 置于休眠模式，该模式停止所有处理器执行（以及芯片上所有其他时钟逻辑）
立即
- 当 GPIO 10 变为高电平时，XOSC 重新启动，程序执行继续

hello_dormant 底层使用 sleep_goto_dormant_until_pin 函数：

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第258至282行

```

258 void sleep_goto_dormant_until_pin(uint gpio_pin, bool edge, bool high) {
259     bool low = !high;
260     bool level = !edge;
261
262     // 在 IO bank 0 配置相应的中断请求 (IRQ)
263     assert(gpio_pin < NUM_BANK0_GPIOS);
264
265     uint32_t event = 0;
266
267     if (level && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_LOW_BITS;
268     if (level && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_LEVEL_HIGH_BITS;
269     if (edge && high) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_HIGH_BITS;
270     if (edge && low) event = IO_BANK0_DORMANT_WAKE_INTE0_GPIO0_EDGE_LOW_BITS;
271
272     gpio_init(gpio_pin);
273     gpio_set_input_enabled(gpio_pin, true);
274     gpio_set_dormant_irq_enabled(gpio_pin, event, true);
275
276     _go_dormant();
277     // 执行在此处停止，直到被唤醒
278
279     // 清除中断请求，以便我们如有需要可再次进入休眠模式
280     gpio_acknowledge_irq(gpio_pin, event);
281     gpio_set_input_enabled(gpio_pin, false);
282 }

```

2.12. 芯片级复位

2.12.1. 概述

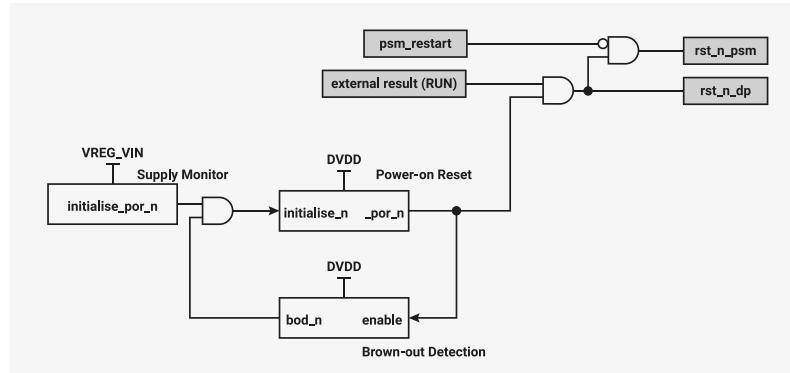
芯片级复位子系统复位整个芯片，使其恢复到默认状态。此操作发生在初次上电时、电源欠压事件期间，或芯片的RUN引脚被拉低时。芯片也可以通过

Rescue Debug Port. See [Section 2.3.4.2, "Rescue DP"](#) for details.

The subsystem has two reset outputs. `rst_n_psm`, which resets the whole chip, except the debug port, and `rst_n_dp`, which only resets the Rescue DP. Both resets are held low at initial power-on, during a brown-out event or when RUN is low. `rst_n_psm` can additionally be held low by the Rescue DP via the subsystem's `psm_restart` input. This allows the chip to be reset via the Rescue DP without resetting the Rescue DP itself. The subsystem releases chip level reset by taking `rst_n_psm` high, handing control to the Power-on State Machine, which continues to start up the chip. See [Section 2.13, "Power-On State Machine"](#) for details.

The chip level reset subsystem is shown in [Figure 21](#), and more information is available in the following sections.

Figure 21. The chip-level reset subsystem

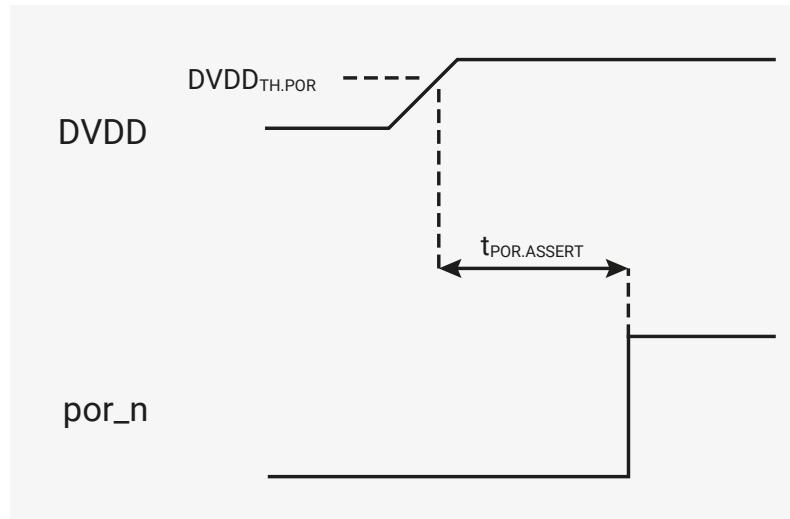


2.12.2. Power-on Reset

The power-on reset block makes sure the chip starts up cleanly when power is first applied by holding it in reset until the digital core supply (`DVDD`) can reliably power the chip's core logic. The block holds its `por_n` output low until `DVDD` has been above the *power-on reset threshold* ($DVDD_{TH.POR}$) for a period greater than the *power-on reset assertion delay* ($t_{POR ASSERT}$). Once high, `por_n` remains high even if `DVDD` subsequently falls below $DVDD_{TH.POR}$, unless brown-out detection is enabled. The behaviour of `por_n` when power is applied is shown in [Figure 22](#).

$DVDD_{TH.POR}$ is fixed at a nominal 0.957V, which should result in a threshold between 0.924V and 0.99V. The threshold assumes a nominal `DVDD` of 1.1V at initial power-on, and `por_n` may never go high if a lower voltage is used. Once the chip is out of reset, `DVDD` can be reduced without `por_n` going low, as long as brown-out detection has been disabled or a suitable threshold voltage has been set.

Figure 22. A power-on reset cycle



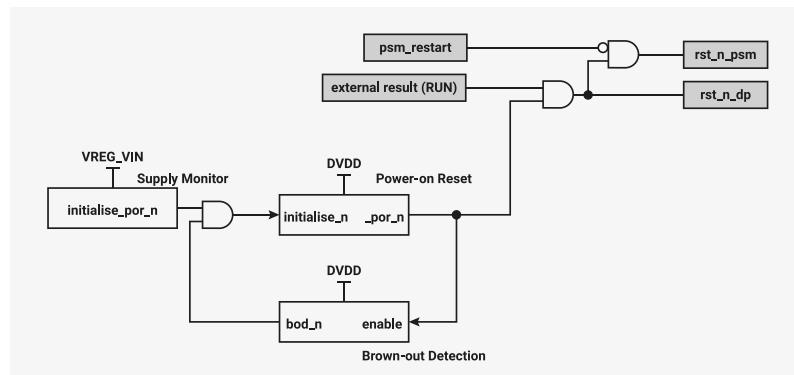
救援调试端口复位。详情请参见第2.3.4.2节，“救援DP”。

该子系统具有两个复位输出。`rst_n_psm`用以复位整个芯片（调试端口除外），`rst_n_dp`仅复位救援DP。两者在初次上电、电源欠压或RUN引脚为低电平时均保持低电平状态。

`rst_n_psm`还可以通过子系统的`psm_restart`输入，由救援DP保持为低电平。该功能允许通过Rescue DP复位芯片，而不复位Rescue DP本身。子系统通过将`rst_n_psm`拉高以释放芯片复位，进而移交控制权给上电状态机，后者继续启动芯片。详情请参见第2.13节，“上电状态机”。

芯片级复位子系统如图21所示，相关信息请见后续章节。

图21。芯片级复位子系统

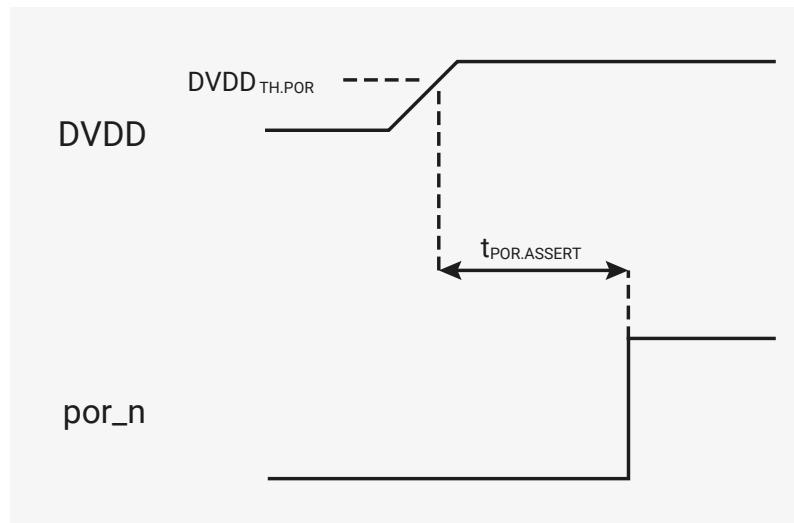


2.12.2. 上电复位

上电复位模块通过保持芯片处于复位状态，直到数字核心电源（DVDD）能够稳定供电给芯片核心逻辑，确保芯片首次上电时的正常启动。该模块将其`por_n`输出保持低电平，直至DVDD高于上电复位阈值（DVDD TH.POR）连续持续超过上电复位保持延时（ $t_{POR ASSERT}$ ）。一旦`por_n`输出变为高电平，即使DVDD随后降至低于DVDD TH.POR，`por_n`仍保持高电平，除非启用了欠压检测功能。加电时`por_n`的行为如图22所示。

DVDD_{TH.POR}固定为标称0.957V，阈值应介于0.924V与0.99V之间。该阈值假定初次加电时DVDD的标称值为1.1V，若使用较低电压，`por_n`可能永远不会置高。芯片解除复位后，只要禁用欠压检测或已设置适当的阈值电压，DVDD可降低而不会导致`por_n`置低。

图22. 上电复位周期



2.12.2.1. Detailed Specifications

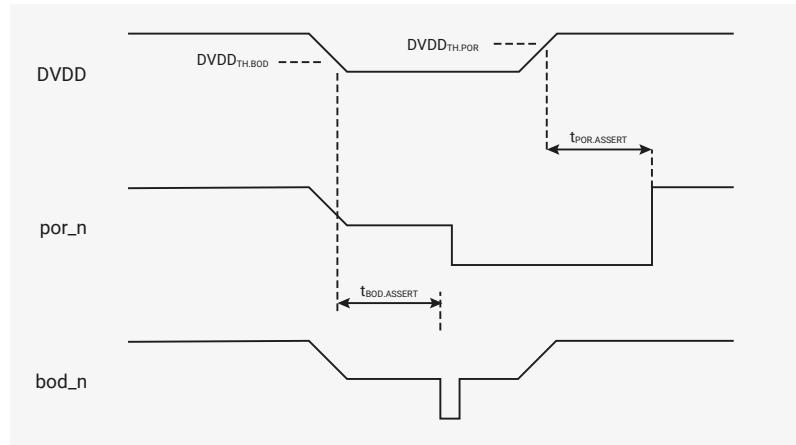
Table 193. Power-on Reset Parameters

Parameter	Description	Min	Typ	Max	Units
DVDD _{TH.POR}	power-on reset threshold	0.924	0.957	0.99	V
t _{POR ASSERT}	power-on reset assertion delay		3	10	μs

2.12.3. Brown-out Detection

The brown-out detection block prevents unreliable operation by initiating a power-on reset cycle if the digital core supply (DVDD) drops below a safe operating level. The block's `bod_n` output is taken low if DVDD drops below the *brown-out detection threshold* (DVDD_{TH.BOD}) for a period longer than the *brown-out detection assertion delay* (t_{BOD ASSERT}). This re-initialises the power-on reset block, which resets the chip, by taking its `por_n` output low, and holds it in reset until DVDD returns to a safe operating level. [Figure 23](#) shows a brown-out event and the subsequent power-on reset cycle.

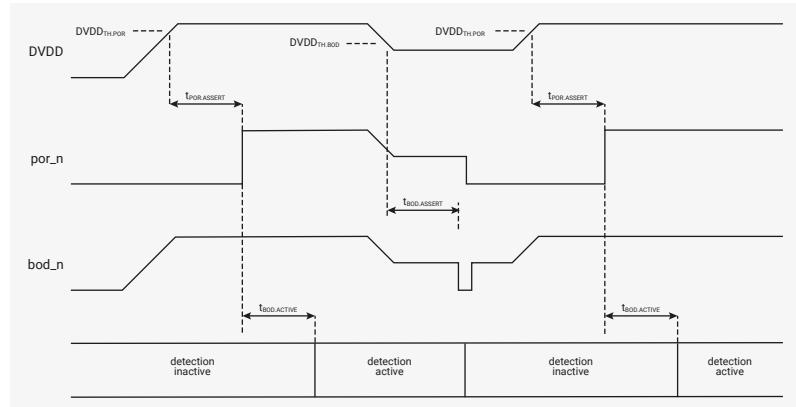
Figure 23. A brown-out detection cycle



2.12.3.1. Detection Enable

Brown-out detection is automatically enabled at initial power-on or after a brown-out initiated reset. There is, however, a short delay, the *brown-out detection activation delay* (t_{BOD ACTIVE}), between `por_n` going high and detection becoming active. This is shown in [Figure 24](#).

Figure 24. Activation of brown-out detection at initial power-on and following a brown-out event.



Once the chip is out of reset, detection can be disabled under software control. This also saves a small amount of power. If detection is subsequently re-enabled, there will be another short delay, the *brown-out detection enable delay* (t_{BOD ENABLE}), before it becomes active again. This is shown in [Figure 25](#).

2.12.2.1. 详细规格

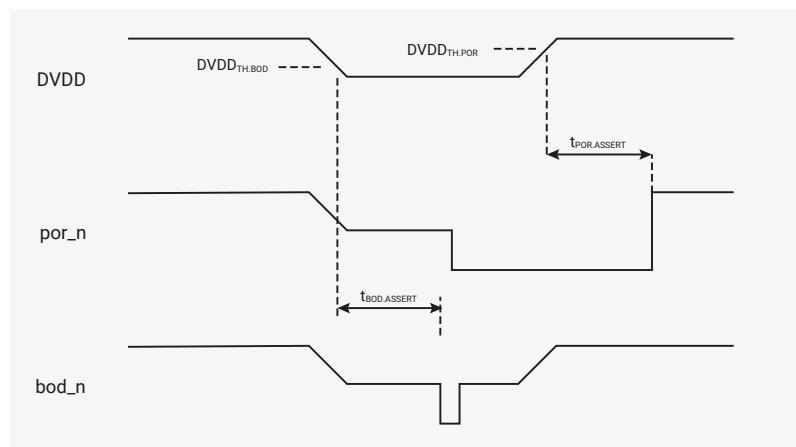
表193。上电复位参数

参数	描述	最小值	典型值	最大值	单位
DVDD _{TH.POR}	上电复位阈值	0.924	0.957	0.99	V
t _{POR ASSERT}	上电复位断言延迟		3	10	μ秒

2.12.3. 欠压检测

欠压检测模块通过在数字核供电电压（DVDD）降至安全工作电平以下时启动加电复位周期，防止不可靠操作。当DVDD降至棕色欠压检测阈值（DVDD_{TH.BOD}）以下且持续时间超过棕色欠压检测断言延迟（t_{BOD ASSERT}）时，模块的bod_n输出将被置低。此操作重新初始化上电复位模块，该模块通过将其por_n输出置低来复位芯片，并持续保持复位状态，直到DVDD恢复至安全工作电平。图23显示了一个棕色欠压事件及其后续的上电复位周期。

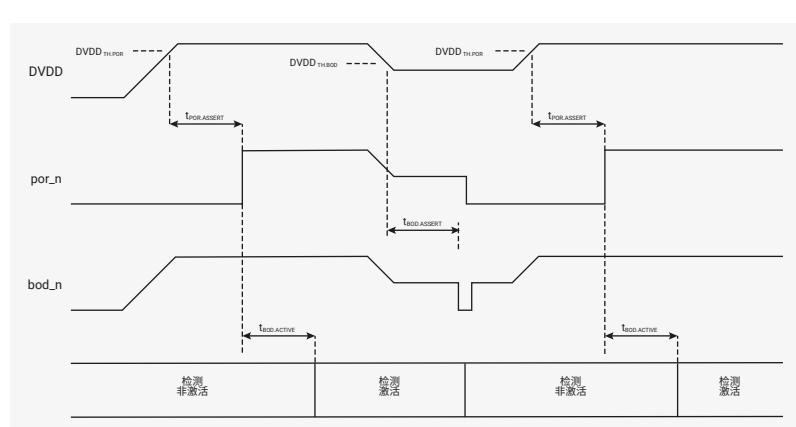
图23。一次棕色欠压检测周期



2.12.3.1. 检测使能

棕色欠压检测在初次上电或因棕色欠压引发的复位后自动启用。然而，在por_n返回高电平与检测启动之间存在短暂延时，即棕色欠压检测激活延迟（t_{BOD ACTIVE}）。如图24所示。

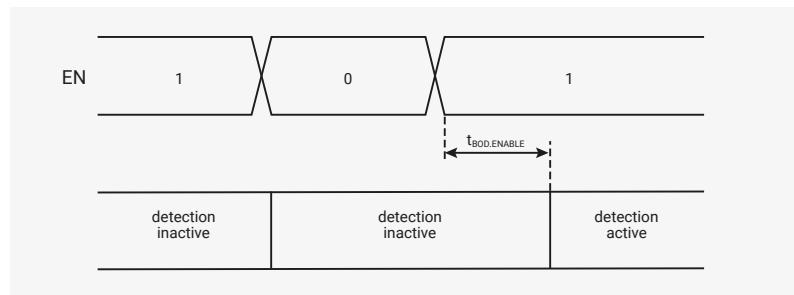
图24。上电初始化及棕褐断电事件后棕褐断电检测的激活。



芯片结束复位后，检测可由软件控制禁用。此举亦可节省少量功耗。若检测随后被重新启用，则在其再次激活之前，会有一段短暂延迟，即棕褐断电检测启用延迟（t_{BOD ENABLE}）。如图25所示。

Detection is disabled by writing a zero to the **EN** field in the **BOD** register and is re-enabled by writing a one to the same field. The block's **bod_n** output is high when detection is disabled.

Figure 25. Disabling and enabling brown-out detection



Detection is re-enabled if the **BOD** register is reset, as this sets the register's **EN** field to one. Again, detection will become active after a delay equal to the *brown-out detection enable delay* ($t_{BOD.ENABLE}$).

NOTE

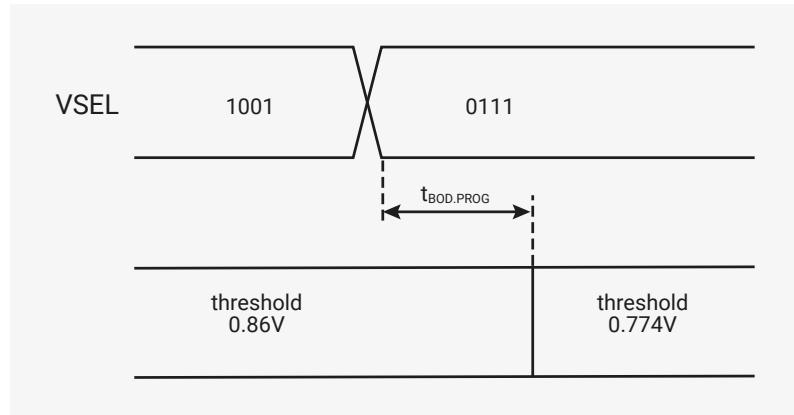
If the **BOD** register is reset by a power-on or brown-out initiated reset, the delay between the register being reset and brown-out detection becoming active will be equal to the *brown-out detection activation delay* ($t_{BOD.ACTIVE}$). The delay will be equal to the *brown-out detection enable delay* ($t_{BOD.ENABLE}$) for all other reset sources.

2.12.3.2. Adjusting the Detection Threshold

The *brown-out detection threshold* ($DVDD_{TH.BOD}$) has a nominal value of 0.86V at initial power-on or after a reset event. This should result in a detection threshold between 0.83V and 0.89V. Once out of reset, the threshold can be adjusted under software control. The new detection threshold will take effect after the *brown-out detection programming delay* ($(t_{BOD.PROG})$. An example of this is shown in Figure 26.

The threshold is adjusted by writing to the **VSEL** field in the **BOD** register. See the **BOD** register description for details.

Figure 26. Adjusting the brown-out detection threshold



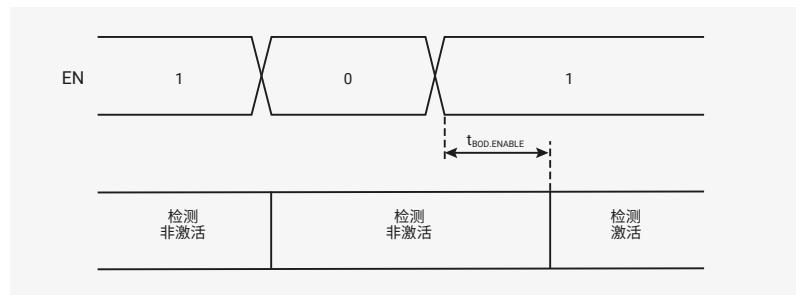
2.12.3.3. Detailed Specifications

Table 194. Brown-out Detection Parameters

Parameter	Description	Min	Typ	Max	Units
$DVDD_{TH.BOD}$	brown-out detection threshold	96.5	100	103.5	% of selected threshold voltage
$t_{BOD.ACTIVE}$	brown-out detection activation delay		55	80	μs

通过向BOD寄存器的EN字段写入0可禁用检测，写入1则可重新启用检测。检测禁用时模块的bod_n输出为高电平。

图25。棕褐断电检测的禁用与启用



BOD寄存器复位时检测将重新启用，因为这会将寄存器的EN字段设置为1。经过等同于棕褐断电检测启用延迟($t_{BOD.ENABLE}$)的延时后，检测将变为激活状态。

i 注意

如果BOD寄存器因上电复位或棕褐断电复位而复位，则从寄存器复位到棕褐断电检测激活之间的延迟等于棕褐断电检测激活延迟($t_{BOD.ACTIVE}$)。对于所有其他复位源，延迟等于棕褐断电检测使能延迟($t_{BOD.ENABLE}$)。

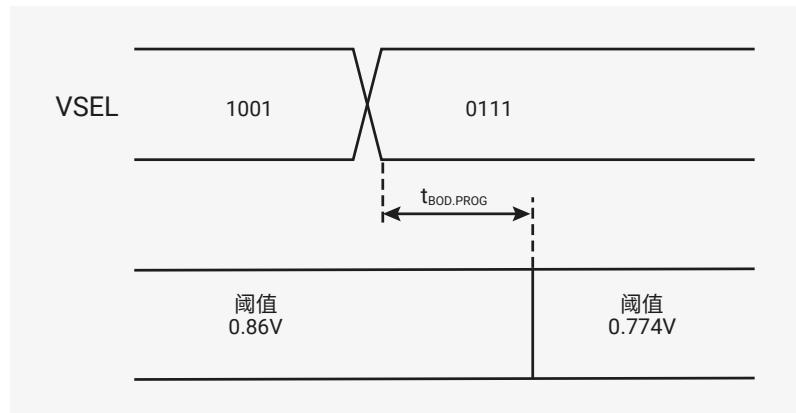
2.12.3.2. 调整检测阈值

棕褐断电检测阈值(DVDDTH.BOD)在初次上电或复位事件后，标称值为0.86V。

此阈值应在0.83V到0.89V之间。复位结束后，阈值可由软件控制进行调整。新的检测阈值将在棕褐断电检测编程延迟($t_{BOD.PROG}$)后生效。图26展示了相关示例。

阈值通过向BOD寄存器中的VSEL字段写入数据来调整。详见BOD寄存器说明。

图26。调整欠压检测阈值



2.12.3.3. 详细规格

表194。欠压检测参数

参数	描述	最小值	典型值	最大值	单位
$DVDD_{TH.BOD}$	欠压检测阈值	96.5	100	103.5	所选阈值电压的百分比
$t_{BOD.ACTIVE}$	欠压检测激活延迟		55	80	μ 秒

Parameter	Description	Min	Typ	Max	Units
$t_{BOD ASSERT}$	brown-out detection assertion delay		3	10	μs
$t_{BOD ENABLE}$	brown-out detection enable delay		35	55	μs
$t_{BOD PROG}$	brown-out detection programming delay		20	30	μs

2.12.4. Supply Monitor

The power-on and brown-out reset blocks are powered by the on-chip voltage regulator's input supply (VREG_VIN). The blocks are initialised when power is first applied, but may not be reliably re-initialised if power is removed and then reapplied before VREG_VIN has dropped to a sufficiently low level. To prevent this happening, VREG_VIN is monitored and the power-on reset block is re-initialised if it drops below the *VREG_VIN activation threshold* ($V_{REG_VIN_{TH ACTIVE}}$). $V_{REG_VIN_{TH ACTIVE}}$ is fixed at a nominal 1.1V, which should result in a threshold between 0.87V and 1.26V. This threshold does not represent a safe operating voltage. It is the voltage that VREG_VIN must drop below to reliably re-initialise the power-on reset block. For safe operation, VREG_VIN must be at a nominal voltage between 1.8V and 3.3V.

2.12.4.1. Detailed Specifications

Table 195. Voltage Regulator Input Supply Monitor Parameters

Parameter	Description	Min	Typ	Max	Units
$V_{REG_VIN_{TH ACTIVE}}$	V_{REG_VIN} activation threshold	0.87	1.1	1.26	V

2.12.5. External Reset

The chip can also be reset by taking its RUN pin low. Taking RUN low will hold the chip in reset irrespective of the state of the core power supply (DVDD) and the power-on reset / brown-out detection blocks. The chip will come out of reset as soon as RUN is taken high, if all other reset sources have been released. RUN can be used to extend the initial power-on reset, or can be driven from an external source to start and stop the chip as required. If RUN is not used, it should be tied high.

2.12.6. Rescue Debug Port Reset

The chip can also be reset via the Rescue Debug Port. This allows the chip to be recovered from a locked up state. In addition to resetting the chip, a Rescue Debug Port reset also sets the `PSM_RESTART_FLAG` in the `CHIP_RESET` register. This is checked by the bootcode at startup, causing it to enter a safe state if the bit is set. See [Section 2.3.4.2, "Rescue DP"](#) for more information.

参数	描述	最小值	典型值	最大值	单位
$t_{BOD ASSERT}$	欠压 检测 断言延迟		3	10	μ秒
$t_{BOD ENABLE}$	欠压 检测启用 延迟		35	55	μ秒
$t_{BOD PROG}$	欠压 检测 编程 延迟		20	30	μ秒

2.12.4. 电源监控

上电和欠压复位模块由片内电压调节器的输入电源（VREG_VIN）供电。模块在首次加电时初始化，但若VREG_VIN尚未降至足够低电平即断电并重新加电，模块可能无法可靠地重新初始化。为防止该情况发生，需监控VREG_VIN电压，当其降至VREG_VIN激活阈值（VREG_VIN_{TH ACTIVE}）以下时，将重新初始化上电复位模块。

VREG_VIN_{TH ACTIVE}固定为名义值1.1V，该阈值范围应介于0.87V至1.26V之间。该阈值并非安全工作电压。该电压为VREG_VIN必须低于的值，以保证可靠地重新初始化上电复位模块。为确保安全运行，VREG_VIN的名义电压须保持在1.8V至3.3V之间。

2.12.4.1. 详细规格

表195。电压调节器
输入电源
监测参数

参数	描述	最小值	典型值	最大值	单位
VREG_VIN _{TH ACTIVE}	VREG_VIN 激活 阈值	0.87	1.1	1.26	V

2.12.5. 外部复位

芯片亦可通过将RUN引脚拉低进行复位。无论核心电源（DVDD）及上电复位/欠压检测模块状态如何，拉低RUN均可使芯片保持复位状态。一旦RUN引脚拉高，且所有其他复位源均已释放，芯片即会退出复位状态。RUN引脚可用于延长初始上电复位时间，或由外部信号驱动，以按需启动和停止芯片。若不使用RUN引脚，应将其固定为高电平。

2.12.6. 救援调试端口复位

芯片亦可通过救援调试端口复位，从而实现从死锁状态恢复。救援调试端口复位不仅复位芯片，同时还会在CHIP_RESET寄存器中设置PSM_RESTART_FLAG标志位。启动时，bootcode会检查该标志位，若被置位则进入安全状态。详情请参见第2.3.4.2节“Rescue DP”。

2.12.7. Source of Last Reset

The source of the most recent chip-level reset can be determined by reading the state of the `HAD_POR`, `HAD_RUN` and `HAD_PSM_RESTART` fields in the `CHIP_RESET` register. A one in the `HAD_POR` field indicates a power supply related reset, i.e. either a power-on or brown-out initiated reset, a one in the `HAD_RUN` field indicates the chip was last reset by the RUN pin, and a one in the `HAD_PSM_RESTART` field indicates the chip has been reset via Rescue Debug Port. There should never be more than one field set to one.

2.12.8. List of Registers

The chip-level reset subsystem shares a register address space with the on-chip voltage regulator. The registers for both subsystems are listed in [Section 2.10.6](#). The shared address space is referred to as `vreg_and_chip_reset` elsewhere in this document.

2.13. Power-On State Machine

2.13.1. Overview

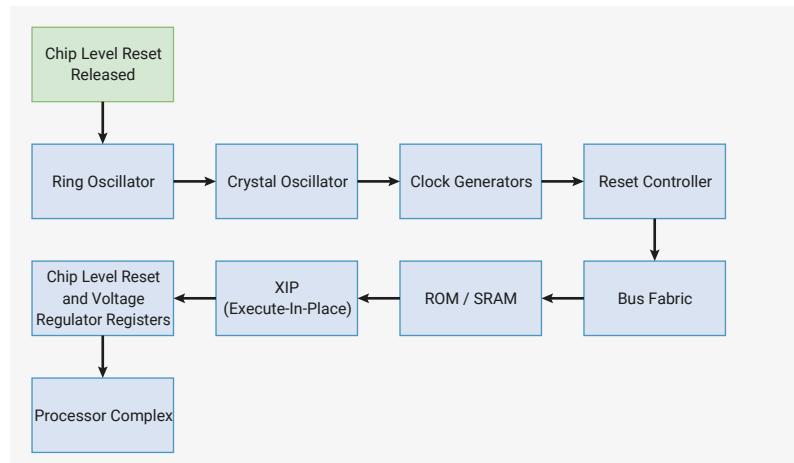
The power-on state machine removes the reset from various hardware blocks in a specific order. Each peripheral in the power-on state machine is controlled by an internal `rst_n` active-low reset signal and generates an internal `rst_done` active-high reset done signal. The power-on state machine deasserts the reset to each peripheral, waits for that peripheral to assert its `rst_done` and then deasserts the reset to the next peripheral. An important use of this is to wait for a clock source to be running cleanly in the chip before the reset to the clock generators is deasserted. This avoids potentially glitchy clocks being distributed to the chip.

The power-on state machine is itself taken out of reset when the Chip-Level Reset subsystem confirms that the digital core supply (DVDD) is powered and stable, and the RUN pin is high. The power-on state machine takes a number of other blocks out of reset at this point via its `rst_n_run` output. This is used to reset things that need to be reset at start-up but must not be reset if the power-on state machine is restarted. This list includes:

- Power on logic in the ring oscillator and crystal oscillator
- Clock dividers which must keep on running during a power-on state machine restart (`clk_ref` and `clk_sys`)
- Watchdog (contains scratch registers which need to persist through a soft-restart of the power-on state machine)

2.13.2. Power On Sequence

Figure 27. Power-On State Machine Sequence.



2.12.7. 最近一次复位来源

最近一次芯片级复位的来源通过读取CHIP_RESET寄存器中 HAD_POR、HAD_RUN和HAD_PSM_RESTART字段状态可得知。HAD_POR字段为1表示电源相关复位，即上电复位或欠压复位；HAD_RUN字段为1表示芯片最近一次复位由RUN引脚触发；HAD_PSM_RESTART字段为1表示芯片通过救援调试端口（Rescue Debug Port）复位。三个字段中不应有多个同时为1。

2.12.8. 寄存器列表

芯片级复位子系统与片上电压调节器共享寄存器地址空间。这两个子系统的寄存器均列于第2.10.6节。文档其他部分将该共享地址空间称为vreg_and_chip_reset。

2.13. 上电状态机

2.13.1. 概述

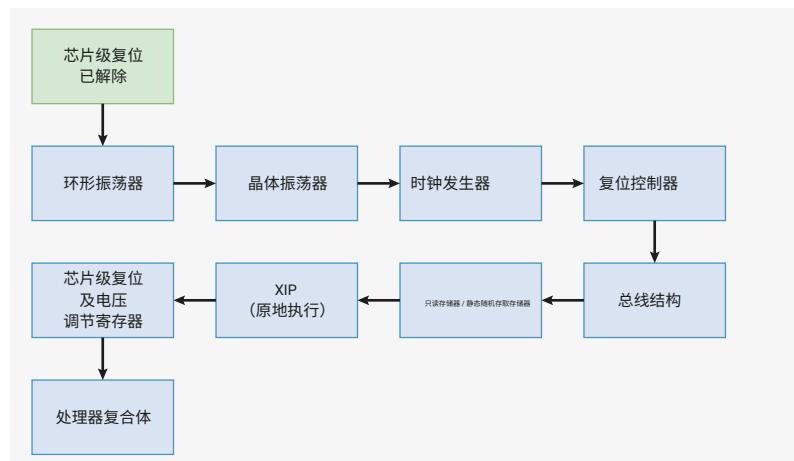
上电状态机按照特定顺序依次解除各硬件模块的复位信号。上电状态机中每个外设均由内部低电平有效复位信号rst_n控制，并产生内部高电平有效复位完成信号rst_done。上电状态机先解除当前外设的复位，等待该外设断言其rst_done信号后，方可解除下一个外设的复位。此机制确保芯片内时钟源稳定运行后，才能解除对时钟发生器的复位。此措施避免了将可能发生故障的时钟分发至芯片。

当芯片级复位子系统确认数字核心电源（DVDD）供电稳定且RUN引脚为高电平时，上电状态机解除复位。此时，上电状态机通过其rst_n_run输出将多个其他模块解除复位。该输出用于复位启动时需复位但在上电状态机重启时不得复位的模块，具体包括：

- 环形振荡器和晶体振荡器中的上电逻辑
- 在上电状态机重启期间必须持续运行的时钟分频器（clk_ref和clk_sys）
- 看门狗（包含需在上电状态机软重启期间保持的暂存寄存器）

2.13.2. 上电序列

图 27. 上电
状态机
序列。



The power-on state machine sequence is as follows:

- Chip-Level Reset subsystem deasserts power-on state machine reset once digital core supply (DVDD) is powered and stable, and `RUN` pin is high (`rst_n_run` is also deasserted at this point)
- Ring Oscillator is started. `rst_done` is asserted once the ripple counter has seen a sufficient number of clock edges to indicate the ring oscillator is stable
- Crystal Oscillator reset is deasserted. The crystal oscillator is not started at this point, so `rst_done` is asserted instantly.
- `clk_ref` and `clk_sys` clock generators are taken out of reset. In the initial configuration `clk_ref` is running from the ring oscillator with no divider. `clk_sys` is running from `clk_ref`. These clocks are needed for the rest of the sequence to progress.

The rest of the sequence is fairly simple, with the following coming out of reset in order one by one:

- Reset Controller - used to reset all non-boot peripherals
- Chip-Level Reset and Voltage Regulator registers - used by the bootrom to check the boot state of the chip. In particular, the `PSM_RESTART_FLAG` flag in the CHIP_RESET register can be set via SWD to indicate to the boot code that there is bad code in flash and it should stop executing. The reset state of the CHIP_RESET register is determined by the Chip-Level Reset subsystem and is not affected by reset coming from the power-on state machine
- XIP (Execute-In-Place) - used by the bootrom to execute code from an external SPI flash
- ROM and SRAM - Boot code is executed from the ROM. SRAM is used by processors and Bus Fabric.
- Bus Fabric - Allows the processors to communicate with peripherals
- Processor complex - Finally the processors can start running

The final thing to come out of reset is the processor complex. This includes both `core0` and `core1`. Both cores will start executing the bootcode from ROM. One of the first things the bootrom does is read the core id. At this point, `core1` will go to sleep leaving `core0` to continue with the bootrom execution. The processor complex has its own reset control and various low-power modes which is why both the `core0` and `core1` resets are deasserted, despite only `core0` being needed for the bootrom.

2.13.3. Register Control

The power-on state machine is a fully automated piece of hardware. It requires no input from the user to work. There are register controls that can be used to override and see the status of the power-on state machine. This allows hardware blocks in the power-on state machine to be reset by software if necessary. There is also a `WDSEL` register which is used to control what is reset by a Watchdog reset.

2.13.4. Interaction with Watchdog

The power-on state machine can be restarted from a software-programmable position if the Watchdog fires. For example, in the case the processor is stuck in an infinite loop, or the programmer has somehow misconfigured the chip. It is important to note that if a peripheral in the power-on state machine has the `WDSEL` bit set, every peripheral after it in the power-on sequence will also be reset because the `rst_done` of the selected peripheral will be deasserted, asserting `rst_n` for the remaining peripherals.

2.13.5. List of Registers

The PSM registers start at a base address of `0x40010000` (defined as `PSM_BASE` in SDK).

上电状态机序列如下：

- 当数字核心电源（DVDD）通电且稳定，且 `RUN`引脚保持高电平（此时 `rst_n_run` 同样解除断言）时，芯片级复位子系统解除上电状态机的复位。
- 启动环形振荡器。当环形计数器检测到足够数量的时钟边沿，表明环形振荡器稳定时，`rst_done`被断言。
- 晶体振荡器复位信号被解除。此时晶体振荡器尚未启动，因此 `rst_done`立即被断言。
- 解除 `clk_ref` 和 `clk_sys` 时钟发生器的复位。在初始配置中，`clk_ref` 由无分频的环形振荡器直接驱动。`clk_sys` 由 `clk_ref` 驱动。这些时钟对于后续序列的执行是必需的。

序列的其余部分较为简洁，复位后以下模块将依次启动：

- 复位控制器——用于复位所有非引导外设
- 芯片级复位及电压调节器寄存器——由引导ROM用于检测芯片的引导状态。特别地，CHIP_RESET寄存器中的`SM_RESTART_FLAG`标志可通过SWD设置，以告知引导代码闪存中存在错误代码，需停止执行。CHIP_RESET寄存器的复位状态由芯片级复位子系统决定，且不受上电状态机复位的影响。
- XIP（Execute-In-Place，原地执行）——引导ROM用于从外部SPI闪存执行代码
- ROM和SRAM——引导代码从ROM执行，SRAM供处理器和总线结构使用。
- 总线结构——使处理器能够与外设通信
- 处理器复合体——处理器最终开始运行

复位结束时，最后启动的是处理器复合体。这包括 `core0` 和 `core1`。两个核心都会开始从ROM执行启动代码。启动ROM执行的首要步骤之一是读取核心ID。此时，`core1` 将进入休眠状态，由 `core0` 继续执行启动ROM。处理器复合体具有自身的复位控制和多种低功耗模式，这就是为什么即使仅需 `core0` 执行启动ROM，`core0` 和 `core1` 的复位仍被解除断言。

2.13.3. 寄存器控制

加电状态机是一种完全自动化的硬件装置。其运行无需用户输入。设有寄存器控制，可用于覆盖并监测加电状态机的状态。如有必要，可通过软件重置加电状态机中的硬件模块。另有一个 `WDSEL` 寄存器，用于控制看门狗复位时被复位的内容。

2.13.4. 与看门狗的交互

当看门狗触发时，可从软件可编程的位置重新启动加电状态机。例如，在处理器陷入无限循环，或程序员错误配置芯片的情况下。

须注意，若电源开启状态机中的某外设设置了 `WDSEL` 位，则电源开启序列中其后续的所有外设亦将被复位，因所选外设的 `rst_done` 信号将被撤销，进而对剩余外设施加复位信号 `rst_n`。

2.13.5. 寄存器列表

PSM寄存器的基地址为 `0x40010000`（在SDK中定义为PSM_BASE）。

Table 196. List of PSM registers

Offset	Name	Info
0x0	FRCE_ON	Force block out of reset (i.e. power it on)
0x4	FRCE_OFF	Force into reset (i.e. power it off)
0x8	WDSEL	Set to 1 if this peripheral should be reset when the watchdog fires.
0xc	DONE	Indicates the peripheral's registers are ready to access.

PSM: FRCE_ON Register

Offset: 0x0

Description

Force block out of reset (i.e. power it on)

Table 197. FRCE_ON Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	PROC1	RW	0x0
15	PROC0	RW	0x0
14	SIO	RW	0x0
13	VREG_AND_CHIP_RESET	RW	0x0
12	XIP	RW	0x0
11	SRAM5	RW	0x0
10	SRAM4	RW	0x0
9	SRAM3	RW	0x0
8	SRAM2	RW	0x0
7	SRAM1	RW	0x0
6	SRAM0	RW	0x0
5	ROM	RW	0x0
4	BUSFABRIC	RW	0x0
3	RESETS	RW	0x0
2	CLOCKS	RW	0x0
1	XOSC	RW	0x0
0	ROSC	RW	0x0

PSM: FRCE_OFF Register

Offset: 0x4

Description

Force into reset (i.e. power it off)

Table 198. FRCE_OFF Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	PROC1	RW	0x0

表196. PSM
寄存器列表

偏移量	名称	说明
0x0	FRCE_ON	强制解除复位（即上电）
0x4	FRCE_OFF	强制进入复位（即断电）
0x8	WDSEL	若该外设在看门狗触发时需复位，则此位设为1。
0xc	DONE	指示该外设寄存器已准备好访问。

PSM: FRCE_ON 寄存器

偏移: 0x0

描述

强制解除复位（即上电）

表197. FRCE_ON
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: FRCE_OFF 寄存器

偏移: 0x4

说明

强制进入复位（即断电）

表198. FRCE_OFF
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0

Bits	Description	Type	Reset
15	PROC0	RW	0x0
14	SIO	RW	0x0
13	VREG_AND_CHIP_RESET	RW	0x0
12	XIP	RW	0x0
11	SRAM5	RW	0x0
10	SRAM4	RW	0x0
9	SRAM3	RW	0x0
8	SRAM2	RW	0x0
7	SRAM1	RW	0x0
6	SRAM0	RW	0x0
5	ROM	RW	0x0
4	BUSFABRIC	RW	0x0
3	RESETS	RW	0x0
2	CLOCKS	RW	0x0
1	XOSC	RW	0x0
0	ROSC	RW	0x0

PSM: WDSEL Register

Offset: 0x8

Description

Set to 1 if this peripheral should be reset when the watchdog fires.

Table 199. WDSEL Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	PROC1	RW	0x0
15	PROC0	RW	0x0
14	SIO	RW	0x0
13	VREG_AND_CHIP_RESET	RW	0x0
12	XIP	RW	0x0
11	SRAM5	RW	0x0
10	SRAM4	RW	0x0
9	SRAM3	RW	0x0
8	SRAM2	RW	0x0
7	SRAM1	RW	0x0
6	SRAM0	RW	0x0
5	ROM	RW	0x0
4	BUSFABRIC	RW	0x0

位	描述	类型	复位值
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: WDSEL 寄存器

偏移: 0x8

描述

若该外设在看门狗触发时需复位，则此位设为1。

表199. WDSEL
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	读写	0x0
15	PROC0	读写	0x0
14	SIO	读写	0x0
13	VREG_AND_CHIP_RESET	读写	0x0
12	XIP	读写	0x0
11	SRAM5	读写	0x0
10	SRAM4	读写	0x0
9	SRAM3	读写	0x0
8	SRAM2	读写	0x0
7	SRAM1	读写	0x0
6	SRAM0	读写	0x0
5	ROM	读写	0x0
4	BUSFABRIC	读写	0x0

Bits	Description	Type	Reset
3	RESETS	RW	0x0
2	CLOCKS	RW	0x0
1	XOSC	RW	0x0
0	ROSC	RW	0x0

PSM: DONE Register

Offset: 0xc

Description

Indicates the peripheral's registers are ready to access.

Table 200. DONE Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	PROC1	RO	0x0
15	PROC0	RO	0x0
14	SIO	RO	0x0
13	VREG_AND_CHIP_RESET	RO	0x0
12	XIP	RO	0x0
11	SRAM5	RO	0x0
10	SRAM4	RO	0x0
9	SRAM3	RO	0x0
8	SRAM2	RO	0x0
7	SRAM1	RO	0x0
6	SRAM0	RO	0x0
5	ROM	RO	0x0
4	BUSFABRIC	RO	0x0
3	RESETS	RO	0x0
2	CLOCKS	RO	0x0
1	XOSC	RO	0x0
0	ROSC	RO	0x0

2.14. Subsystem Resets

2.14.1. Overview

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in RP2040. This includes:

- USB Controller

位	描述	类型	复位值
3	RESETS	读写	0x0
2	CLOCKS	读写	0x0
1	XOSC	读写	0x0
0	ROSC	读写	0x0

PSM: DONE 寄存器

偏移: 0xc

描述

指示该外设寄存器已准备好访问。

表 200. 完成寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	PROC1	只读	0x0
15	PROC0	只读	0x0
14	SIO	只读	0x0
13	VREG_AND_CHIP_RESET	只读	0x0
12	XIP	只读	0x0
11	SRAM5	只读	0x0
10	SRAM4	只读	0x0
9	SRAM3	只读	0x0
8	SRAM2	只读	0x0
7	SRAM1	只读	0x0
6	SRAM0	只读	0x0
5	ROM	只读	0x0
4	BUSFABRIC	只读	0x0
3	RESETS	只读	0x0
2	CLOCKS	只读	0x0
1	XOSC	只读	0x0
0	ROSC	只读	0x0

2.14. 子系统复位

2.14.1. 概述

复位控制器允许软件控制对 RP2040 中所有非关键启动外设的复位，包括：

- USB 控制器

- PIO
- Peripherals such as UART, I2C, SPI, PWM, Timer, ADC
- PLLs
- IO and Pad registers

The full list can be seen in the register descriptions.

Every peripheral reset by the reset controller is held in reset at power-up. It is up to software to deassert the reset of peripherals it intends to use. Note that if you are using the SDK some peripherals may already be out of reset.

2.14.2. Programmer's Model

The SDK defines a struct to represent the resets registers.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/resets.h Lines 59 - 146

```

59 typedef struct {
60     _REG_(RESETS_RESET_OFFSET) // RESETS_RESET
61     // Reset control.
62     // 0x01000000 [24]    USBCTRL      (1)
63     // 0x00800000 [23]    UART1        (1)
64     // 0x00400000 [22]    UART0        (1)
65     // 0x00200000 [21]    TIMER         (1)
66     // 0x00100000 [20]    TBMAN         (1)
67     // 0x00080000 [19]    SYSINFO       (1)
68     // 0x00040000 [18]    SYSCFG        (1)
69     // 0x00020000 [17]    SPI1          (1)
70     // 0x00010000 [16]    SPI0          (1)
71     // 0x00008000 [15]    RTC           (1)
72     // 0x00004000 [14]    PWM           (1)
73     // 0x00002000 [13]    PLL_USB        (1)
74     // 0x00001000 [12]    PLL_SYS        (1)
75     // 0x00000800 [11]    PIO1          (1)
76     // 0x00000400 [10]    PIO0          (1)
77     // 0x00000200 [9]     PADS_QSPI     (1)
78     // 0x00000100 [8]     PADS_BANK0    (1)
79     // 0x00000080 [7]     JTAG           (1)
80     // 0x00000040 [6]     IO_QSPI       (1)
81     // 0x00000020 [5]     IO_BANK0      (1)
82     // 0x00000010 [4]     I2C1          (1)
83     // 0x00000008 [3]     I2C0          (1)
84     // 0x00000004 [2]     DMA            (1)
85     // 0x00000002 [1]     BUSCTRL       (1)
86     // 0x00000001 [0]     ADC            (1)
87     io_rw_32 reset;
88
89     _REG_(RESETS_WDSEL_OFFSET) // RESETS_WDSEL
90     // Watchdog select.
91     // 0x01000000 [24]    USBCTRL      (0)
92     // 0x00800000 [23]    UART1        (0)
93     // 0x00400000 [22]    UART0        (0)
94     // 0x00200000 [21]    TIMER         (0)
95     // 0x00100000 [20]    TBMAN         (0)
96     // 0x00080000 [19]    SYSINFO       (0)
97     // 0x00040000 [18]    SYSCFG        (0)
98     // 0x00020000 [17]    SPI1          (0)
99     // 0x00010000 [16]    SPI0          (0)
100    // 0x00008000 [15]    RTC           (0)
101    // 0x00004000 [14]    PWM           (0)
102    // 0x00002000 [13]    PLL_USB        (0)

```

- PIO
- UART、I2C、SPI、PWM、定时器、ADC 等外设
- 锁相环（PLL）
- IO 与引脚寄存器

完整列表详见寄存器说明。

由复位控制器复位的每个外设均在上电时保持复位状态。软件需负责取消其计划使用的外设的复位。请注意，使用 SDK 时，部分外设可能已解除复位。

2.14.2. 程序员模型

SDK 定义了表示复位寄存器的结构体。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/resets.h 第 59 至 146 行

```

59 typedef struct {
60     _REG_(RESETS_RESET_OFFSET) // RESETS_RESET
61     //重置控制。
62     // 0x01000000 [24]    USBCTRL      (1)
63     // 0x00800000 [23]    UART1        (1)
64     // 0x00400000 [22]    UART0        (1)
65     // 0x00200000 [21]    TIMER         (1)
66     // 0x00100000 [20]    TBMAN         (1)
67     // 0x00080000 [19]    SYSINFO       (1)
68     // 0x00040000 [18]    SYSCFG        (1)
69     // 0x00020000 [17]    SPI1          (1)
70     // 0x00010000 [16]    SPI0          (1)
71     // 0x00008000 [15]    RTC           (1)
72     // 0x00004000 [14]    PWM           (1)
73     // 0x00002000 [13]    PLL_USB        (1)
74     // 0x00001000 [12]    PLL_SYS        (1)
75     // 0x00000800 [11]    PIO1          (1)
76     // 0x00000400 [10]    PIO0          (1)
77     // 0x00000200 [9]     PADS_QSPI     (1)
78     // 0x00000100 [8]     PADS_BANK0    (1)
79     // 0x00000080 [7]     JTAG           (1)
80     // 0x00000040 [6]     IO_QSPI       (1)
81     // 0x00000020 [5]     IO_BANK0      (1)
82     // 0x00000010 [4]     I2C1          (1)
83     // 0x00000008 [3]     I2C0          (1)
84     // 0x00000004 [2]     DMA            (1)
85     // 0x00000002 [1]     BUSCTRL       (1)
86     // 0x00000001 [0]     ADC            (1)
87     io_rw_32 reset;
88
89     _REG_(RESETS_WDSEL_OFFSET) // RESETS_WDSEL
90     //看门狗选择
91     // 0x01000000 [24]    USBCTRL      (0)
92     // 0x00800000 [23]    UART1        (0)
93     // 0x00400000 [22]    UART0        (0)
94     // 0x00200000 [21]    TIMER         (0)
95     // 0x00100000 [20]    TBMAN         (0)
96     // 0x00080000 [19]    SYSINFO       (0)
97     // 0x00040000 [18]    SYSCFG        (0)
98     // 0x00020000 [17]    SPI1          (0)
99     // 0x00010000 [16]    SPI0          (0)
100    // 0x00008000 [15]    RTC           (0)
101    // 0x00004000 [14]    PWM           (0)
102    // 0x00002000 [13]    PLL_USB        (0)

```

```

103 // 0x000001000 [12] PLL_SYS (0)
104 // 0x000000800 [11] PIO1 (0)
105 // 0x00000400 [10] PIO0 (0)
106 // 0x00000200 [9] PADS_QSPI (0)
107 // 0x00000100 [8] PADS_BANK0 (0)
108 // 0x00000080 [7] JTAG (0)
109 // 0x00000040 [6] IO_QSPI (0)
110 // 0x00000020 [5] IO_BANK0 (0)
111 // 0x00000010 [4] I2C1 (0)
112 // 0x00000008 [3] I2C0 (0)
113 // 0x00000004 [2] DMA (0)
114 // 0x00000002 [1] BUSCTRL (0)
115 // 0x00000001 [0] ADC (0)
116 io_rw_32 wdsel;
117
118 _REG_(RESETS_RESET_DONE_OFFSET) // RESETS_RESET_DONE
119 // Reset done.
120 // 0x01000000 [24] USBCTRL (0)
121 // 0x00800000 [23] UART1 (0)
122 // 0x00400000 [22] UART0 (0)
123 // 0x00200000 [21] TIMER (0)
124 // 0x00100000 [20] TBMAN (0)
125 // 0x00080000 [19] SYSINFO (0)
126 // 0x00040000 [18] SYSCFG (0)
127 // 0x00020000 [17] SPI1 (0)
128 // 0x00010000 [16] SPI0 (0)
129 // 0x00008000 [15] RTC (0)
130 // 0x00004000 [14] PWM (0)
131 // 0x00002000 [13] PLL_USB (0)
132 // 0x00001000 [12] PLL_SYS (0)
133 // 0x00000800 [11] PIO1 (0)
134 // 0x00000400 [10] PIO0 (0)
135 // 0x00000200 [9] PADS_QSPI (0)
136 // 0x00000100 [8] PADS_BANK0 (0)
137 // 0x00000080 [7] JTAG (0)
138 // 0x00000040 [6] IO_QSPI (0)
139 // 0x00000020 [5] IO_BANK0 (0)
140 // 0x00000010 [4] I2C1 (0)
141 // 0x00000008 [3] I2C0 (0)
142 // 0x00000004 [2] DMA (0)
143 // 0x00000002 [1] BUSCTRL (0)
144 // 0x00000001 [0] ADC (0)
145 io_ro_32 reset_done;
146 } resets_hw_t;

```

Three registers are defined:

- **reset**: this register contains a bit for each peripheral that can be reset. If the bit is set to 1 then the reset is asserted. If the bit is cleared then the reset is deasserted.
- **wdsel**: if the bit is set then this peripheral will be reset if the watchdog fires (note that the power on state machine can potentially reset the whole reset controller, which will reset everything)
- **reset_done**: a bit for each peripheral, that gets set once the peripheral is out of reset. This allows software to wait for this status bit in case the peripheral has some initialisation to do before it can be used.

The reset functions in the SDK are defined as follows:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h Lines 121 - 123

```

121 static __force_inline void reset_block(uint32_t bits) {
122     reset_block_mask(bits);

```

```

103 // 0x00001000 [12]    PLL_SYS      (0)
104 // 0x00000800 [11]    PIO1        (0)
105 // 0x00000400 [10]    PIO0        (0)
106 // 0x00000200 [9]     PADS_QSPI   (0)
107 // 0x00000100 [8]     PADS_BANK0 (0)
108 // 0x00000080 [7]     JTAG         (0)
109 // 0x00000040 [6]     IO_QSPI    (0)
110 // 0x00000020 [5]     IO_BANK0  (0)
111 // 0x00000010 [4]     I2C1        (0)
112 // 0x00000008 [3]     I2C0        (0)
113 // 0x00000004 [2]     DMA          (0)
114 // 0x00000002 [1]     BUSCTRL    (0)
115 // 0x00000001 [0]     ADC          (0)
116 io_rw_32 wdsel;
117
118 _REG_(RESETS_RESET_DONE_OFFSET) // RESETS_RESET_DONE
119 // 复位完成。
120 // 0x01000000 [24]    USBCTRL    (0)
121 // 0x00800000 [23]    UART1      (0)
122 // 0x00400000 [22]    UART0      (0)
123 // 0x00200000 [21]    TIMER       (0)
124 // 0x00100000 [20]    TBMAN      (0)
125 // 0x00080000 [19]    SYSINFO    (0)
126 // 0x00040000 [18]    SYSCFG     (0)
127 // 0x00020000 [17]    SPI1        (0)
128 // 0x00010000 [16]    SPI0        (0)
129 // 0x00008000 [15]    RTC         (0)
130 // 0x00004000 [14]    PWM         (0)
131 // 0x00002000 [13]    PLL_USB    (0)
132 // 0x00001000 [12]    PLL_SYS    (0)
133 // 0x00000800 [11]    PIO1        (0)
134 // 0x00000400 [10]    PIO0        (0)
135 // 0x00000200 [9]     PADS_QSPI  (0)
136 // 0x00000100 [8]     PADS_BANK0 (0)
137 // 0x00000080 [7]     JTAG         (0)
138 // 0x00000040 [6]     IO_QSPI    (0)
139 // 0x00000020 [5]     IO_BANK0  (0)
140 // 0x00000010 [4]     I2C1        (0)
141 // 0x00000008 [3]     I2C0        (0)
142 // 0x00000004 [2]     DMA          (0)
143 // 0x00000002 [1]     BUSCTRL    (0)
144 // 0x00000001 [0]     ADC          (0)
145 io_ro_32 reset_done;
146 } resets_hw_t;

```

定义了三个寄存器：

- **reset**: 该寄存器包含针对每个可复位外设的位。如果该位被设置为 1，则复位被激活。
如果该位被清除，则复位被解除。
- **wdsel**: 若设置该位，则在看门狗触发时，该外设将被复位（注意，电源上电状态机可能复位整个复位控制器，从而复位所有外设）。
- **reset_done**: 每个外设对应一个位，外设完成复位后该位被置位。这允许软件等待该状态位，以确保外设完成初始化后方可使用。

SDK 中复位功能定义如下：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第121至123行

```

121 static __force_inline void reset_block(uint32_t bits) {
122     reset_block_mask(bits);

```

```
123 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h Lines 125 - 127

```
125 static __force_inline void unreset_block(uint32_t bits) {
126     unreset_block_mask(bits);
127 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h Lines 129 - 131

```
129 static __force_inline void unreset_block_wait(uint32_t bits) {
130     return unreset_block_mask_wait_blocking(bits);
131 }
```

An example use of these is in the UART driver, where the driver defines a `uart_reset` function, selecting a different bit of the reset register depending on the uart specified:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c Lines 32 - 38

```
32 static inline void uart_reset(uart_inst_t *uart) {
33     reset_block_num(uart_get_reset_num(uart));
34 }
35
36 static inline void uart_unreset(uart_inst_t *uart) {
37     unreset_block_num_wait_blocking(uart_get_reset_num(uart));
38 }
```

2.14.3. List of Registers

The reset controller registers start at a base address of `0x4000c000` (defined as `RESETS_BASE` in SDK).

Table 201. List of RESETS registers

Offset	Name	Info
0x0	RESET	Reset control.
0x4	WDSEL	Watchdog select.
0x8	RESET_DONE	Reset done.

RESETS: RESET Register

Offset: 0x0

Description

Reset control. If a bit is set it means the peripheral is in reset. 0 means the peripheral's reset is deasserted.

Table 202. RESET Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24	USBCTRL	RW	0x1
23	UART1	RW	0x1
22	UART0	RW	0x1
21	TIMER	RW	0x1

```
123 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第125至127行

```
125 static __force_inline void unreset_block(uint32_t bits) {
126     unreset_block_mask(bits);
127 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_resets/include/hardware/resets.h 第129至131行

```
129 static __force_inline void unreset_block_wait(uint32_t bits) {
130     return unreset_block_mask_wait_blocking(bits);
131 }
```

这些用法的一个示例出现在UART驱动中，驱动根据指定的uart选择复位寄存器的不同位，定义了uart_reset函数：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c 第32至38行

```
32 static inline void uart_reset(uart_inst_t *uart) {
33     reset_block_num(uart_get_reset_num(uart));
34 }
35
36 static inline void uart_unreset(uart_inst_t *uart) {
37     unreset_block_num_wait_blocking(uart_get_reset_num(uart));
38 }
```

2.14.3. 寄存器列表

复位控制器寄存器基地址为 **0x4000c000**（在SDK中定义为RESETS_BASE）。

表201.
RESETS寄存器列表

偏移量	名称	说明
0x0	RESET	复位控制。
0x4	WDSEL	看门狗选择。
0x8	RESET_DONE	复位完成。

RESETS：RESET寄存器

偏移: 0x0

描述

复位控制。位被置位表示外围设备处于复位状态；0表示外围设备的复位已解除。

表202. RESET
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	读写	0x1
23	UART1	读写	0x1
22	UART0	读写	0x1
21	TIMER	读写	0x1

Bits	Description	Type	Reset
20	TBMAN	RW	0x1
19	SYSINFO	RW	0x1
18	SYSCFG	RW	0x1
17	SPI1	RW	0x1
16	SPI0	RW	0x1
15	RTC	RW	0x1
14	PWM	RW	0x1
13	PLL_USB	RW	0x1
12	PLL_SYS	RW	0x1
11	PIO1	RW	0x1
10	PIO0	RW	0x1
9	PADS_QSPI	RW	0x1
8	PADS_BANK0	RW	0x1
7	JTAG	RW	0x1
6	IO_QSPI	RW	0x1
5	IO_BANK0	RW	0x1
4	I2C1	RW	0x1
3	I2C0	RW	0x1
2	DMA	RW	0x1
1	BUSCTRL	RW	0x1
0	ADC	RW	0x1

RESETS: WDSEL Register

Offset: 0x4

Description

Watchdog select. If a bit is set then the watchdog will reset this peripheral when the watchdog fires.

Table 203. WDSEL Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24	USBCTRL	RW	0x0
23	UART1	RW	0x0
22	UART0	RW	0x0
21	TIMER	RW	0x0
20	TBMAN	RW	0x0
19	SYSINFO	RW	0x0
18	SYSCFG	RW	0x0
17	SPI1	RW	0x0

位	描述	类型	复位值
20	TBMAN	读写	0x1
19	SYSINFO	读写	0x1
18	SYSCFG	读写	0x1
17	SPI1	读写	0x1
16	SPI0	读写	0x1
15	RTC	读写	0x1
14	PWM	读写	0x1
13	PLL_USB	读写	0x1
12	PLL_SYS	读写	0x1
11	PIO1	读写	0x1
10	PIO0	读写	0x1
9	PADS_QSPI	读写	0x1
8	PADS_BANK0	读写	0x1
7	JTAG	读写	0x1
6	IO_QSPI	读写	0x1
5	IO_BANK0	读写	0x1
4	I2C1	读写	0x1
3	I2C0	读写	0x1
2	DMA	读写	0x1
1	BUSCTRL	读写	0x1
0	ADC	读写	0x1

复位：WDSEL寄存器

偏移：0x4

说明

看门狗选择。若某位被置位，则当看门狗触发时，该看门狗将重置对应外设。

表 203. WDSEL
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	读写	0x0
23	UART1	读写	0x0
22	UART0	读写	0x0
21	TIMER	读写	0x0
20	TBMAN	读写	0x0
19	SYSINFO	读写	0x0
18	SYSCFG	读写	0x0
17	SPI1	读写	0x0

Bits	Description	Type	Reset
16	SPI0	RW	0x0
15	RTC	RW	0x0
14	PWM	RW	0x0
13	PLL_USB	RW	0x0
12	PLL_SYS	RW	0x0
11	PIO1	RW	0x0
10	PIO0	RW	0x0
9	PADS_QSPI	RW	0x0
8	PADS_BANK0	RW	0x0
7	JTAG	RW	0x0
6	IO_QSPI	RW	0x0
5	IO_BANK0	RW	0x0
4	I2C1	RW	0x0
3	I2C0	RW	0x0
2	DMA	RW	0x0
1	BUSCTRL	RW	0x0
0	ADC	RW	0x0

RESETS: RESET_DONE Register

Offset: 0x8

Description

Reset done. If a bit is set then a reset done signal has been returned by the peripheral. This indicates that the peripheral's registers are ready to be accessed.

Table 204.
RESET_DONE Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24	USBCTRL	RO	0x0
23	UART1	RO	0x0
22	UART0	RO	0x0
21	TIMER	RO	0x0
20	TBMAN	RO	0x0
19	SYSINFO	RO	0x0
18	SYSCFG	RO	0x0
17	SPI1	RO	0x0
16	SPI0	RO	0x0
15	RTC	RO	0x0
14	PWM	RO	0x0
13	PLL_USB	RO	0x0

位	描述	类型	复位值
16	SPI0	读写	0x0
15	RTC	读写	0x0
14	PWM	读写	0x0
13	PLL_USB	读写	0x0
12	PLL_SYS	读写	0x0
11	PIO1	读写	0x0
10	PIO0	读写	0x0
9	PADS_QSPI	读写	0x0
8	PADS_BANK0	读写	0x0
7	JTAG	读写	0x0
6	IO_QSPI	读写	0x0
5	IO_BANK0	读写	0x0
4	I2C1	读写	0x0
3	I2C0	读写	0x0
2	DMA	读写	0x0
1	BUSCTRL	读写	0x0
0	ADC	读写	0x0

重置：RESET_DONE 寄存器

偏移: 0x8

描述

重置完成。若某位被置位，表明外设已返回重置完成信号，指示外设寄存器已准备好访问。

表 204。
RESET_DONE 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	USBCTRL	只读	0x0
23	UART1	只读	0x0
22	UART0	只读	0x0
21	TIMER	只读	0x0
20	TBMAN	只读	0x0
19	SYSINFO	只读	0x0
18	SYSCFG	只读	0x0
17	SPI1	只读	0x0
16	SPI0	只读	0x0
15	RTC	只读	0x0
14	PWM	只读	0x0
13	PLL_USB	只读	0x0

Bits	Description	Type	Reset
12	PLL_SYS	RO	0x0
11	PIO1	RO	0x0
10	PIO0	RO	0x0
9	PADS_QSPI	RO	0x0
8	PADS_BANK0	RO	0x0
7	JTAG	RO	0x0
6	IO_QSPI	RO	0x0
5	IO_BANK0	RO	0x0
4	I2C1	RO	0x0
3	I2C0	RO	0x0
2	DMA	RO	0x0
1	BUSCTRL	RO	0x0
0	ADC	RO	0x0

2.15. Clocks

2.15.1. Overview

The clocks block provides independent clocks to on-chip and external components. It takes inputs from a variety of clock sources allowing the user to trade off performance against cost, board area and power consumption. From these sources it uses multiple clock generators to provide the required clocks. This architecture allows the user flexibility to start and stop clocks independently and to vary some clock frequencies whilst maintaining others at their optimum frequencies.

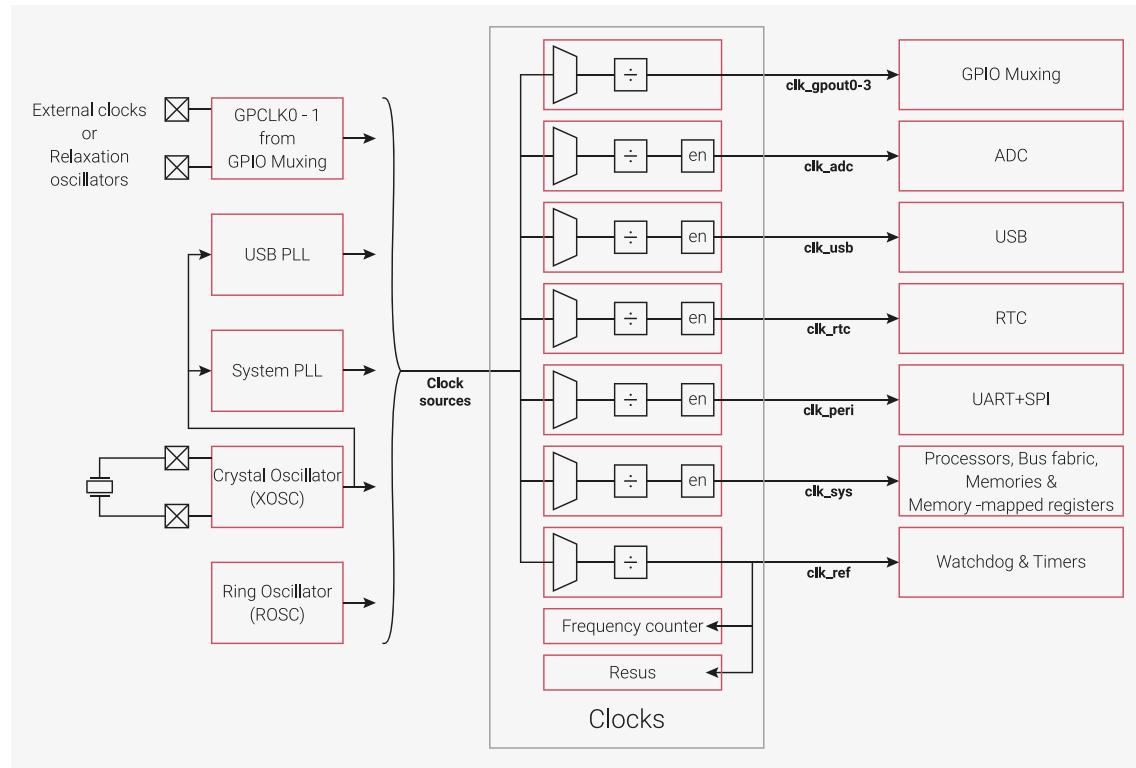
位	描述	类型	复位值
12	PLL_SYS	只读	0x0
11	PIO1	只读	0x0
10	PIO0	只读	0x0
9	PADS_QSPI	只读	0x0
8	PADS_BANK0	只读	0x0
7	JTAG	只读	0x0
6	IO_QSPI	只读	0x0
5	IO_BANK0	只读	0x0
4	I2C1	只读	0x0
3	I2C0	只读	0x0
2	DMA	只读	0x0
1	BUSCTRL	只读	0x0
0	ADC	只读	0x0

2.15. 时钟

2.15.1. 概述

时钟模块为片上及外部组件提供独立时钟。其接受多种时钟源输入，允许用户在性能、成本、板载面积及功耗间进行权衡。该模块利用多个时钟发生器，从这些时钟源产生所需时钟。该架构赋予用户灵活性，可独立启停时钟，并在保持部分时钟频率最优化的同时调整其他时钟频率。

Figure 28. Clocks overview



For very low cost or low power applications where precise timing is not required, the chip can be run from the internal Ring Oscillator (ROSC). Alternatively the user can provide external clocks or construct simple relaxation oscillators using the GPIOs and appropriate external passive components. Where timing is more critical, the Crystal Oscillator (XOSC) can provide an accurate reference to the 2 on-chip PLLs to provide fast clocking at precise frequencies.

The clock generators select from the clock sources and optionally divide the selected clock before outputting through enable logic which provides automatic clock disabling in SLEEP mode (see [Section 2.11.2](#)).

An on-chip frequency counter facilitates debugging of the clock setup and also allows measurement of the frequencies of external clocks. The on-chip resus component restarts the system clock from a known good clock if it is accidentally stopped. This allows the software debugger to access registers and debug the problem.

The chip has an ultra-low power mode called DORMANT (see [Section 2.11.3](#)) in which all on-chip clock sources are stopped to save power. External sources are not stopped and can be used to provide a clock to the on-chip RTC which can provide an alarm to wake the chip from DORMANT mode. Alternatively the GPIO interrupts can be configured to wake the chip from DORMANT mode in response to an external event.

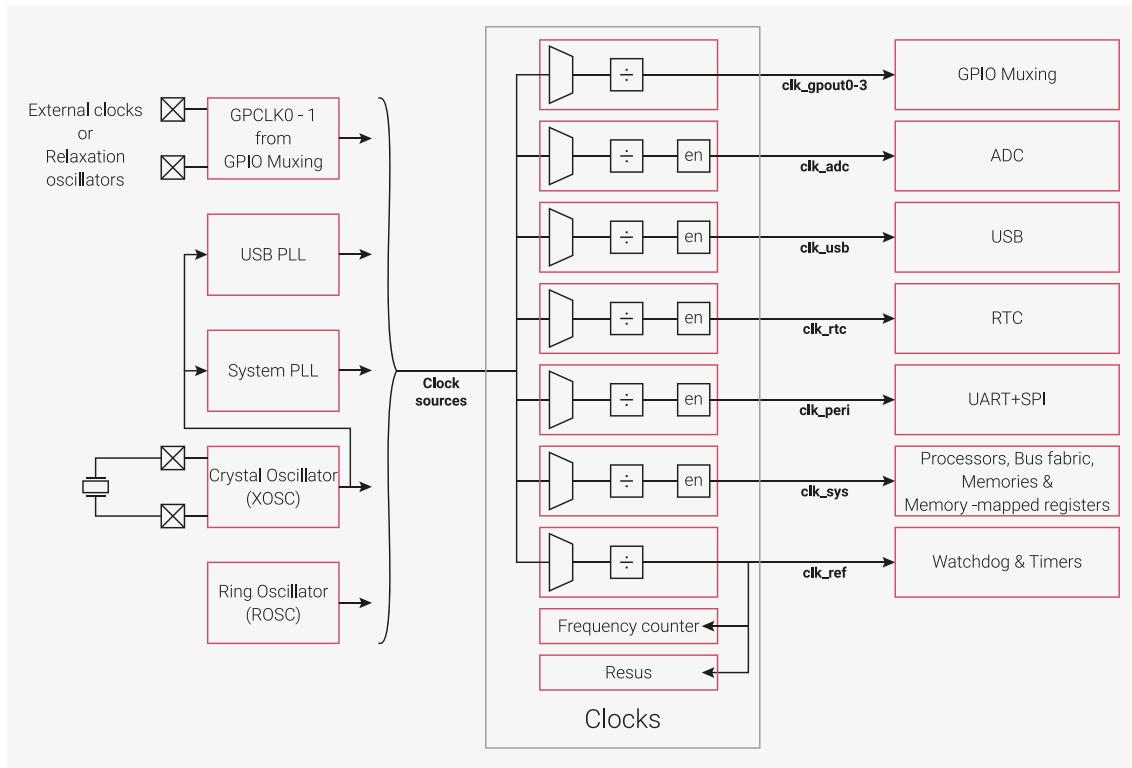
Up to 4 generated clocks can be output to GPIOs at up to 50MHz. This allows the user to supply clocks to external devices, thus reducing component counts in power, space and cost sensitive applications.

2.15.2. Clock sources

The RP2040 can be run from a variety of clock sources. This flexibility allows the user to optimise the clock setup for performance, cost, board area and power consumption. The sources include the on-chip Ring Oscillator ([Section 2.17](#)), the Crystal Oscillator ([Section 2.16](#)), external clocks from GPIOs ([Section 2.15.6.4](#)) and the PLLs ([Section 2.18](#)).

The list of clock sources is different per clock generator and can be found as enumerated values in the `CTRL` register. See `CLK_SYS_CTRL` as an example.

图 28. 时钟
概述



对于不要求精确时序的极低成本或低功耗应用，芯片可由内部环形振荡器（ROSC）驱动。或者用户可提供外部时钟，亦可利用GPIO及适当的外部无源元件构建简单的回馈振荡器。在时序要求更为严格的情况下，晶体振荡器（XOSC）可为两个片内锁相环（PLL）提供精准参考，以实现快速且频率精确的时钟。

时钟发生器从时钟源中选择时钟信号，并可对选定时钟进行分频，随后通过使能逻辑输出，该逻辑在休眠模式（参见第2.11.2节）下提供自动时钟禁用功能。

片内频率计数器便于调试时钟设置，同时可测量外部时钟频率。片内复位组件在系统时钟意外停止时，能够从已知的正常时钟重新启动系统时钟。此功能使软件调试器能够访问寄存器并进行问题调试。

芯片设有名为休眠模式（参见第2.11.3节）的超低功耗模式，所有片内时钟源均被停止，以节省能耗。外部时钟源不会被停止，可用于为片内实时时钟（RTC）提供时钟，RTC能触发报警以唤醒芯片脱离休眠模式。GPIO中断亦可配置为响应外部事件，从休眠模式中唤醒芯片。

最多可将4个产生的时钟以最高50MHz的频率输出至GPIO。该功能允许用户向外部设备提供时钟，从而在对功率、空间和成本敏感的应用中减少组件数量。

2.15.2. 时钟源

RP2040可由多种时钟源提供时钟。此灵活性使用户能够针对性能、成本、电路板面积及功耗优化时钟配置。时钟源包括片上环形振荡器（第2.17节）、晶体振荡器（第2.16节）、来自GPIO的外部时钟（第2.15.6.4节）以及锁相环（第2.18节）。

各时钟发生器支持的时钟源列表不同，详见 [CTRL](#) 寄存器中的枚举值。

参见 [CLK_SYS_CTRL](#) 作为示例。

2.15.2.1. Ring Oscillator

The on-chip Ring Oscillator ([Section 2.17](#)) requires no external components. It runs automatically from power-up and is used to clock the chip during the initial boot stages. The startup frequency is typically 6MHz but varies with PVT (Process, Voltage and Temperature). The frequency is likely to be in the range 4-8MHz and is guaranteed to be in the range 1.8-12MHz.

For low cost applications where frequency accuracy is unimportant, the chip can continue to run from the ROSC. If greater performance is required the frequency can be increased by programming the registers as described in [Section 2.17](#). The frequency will vary with PVT (Process, Voltage and Temperature) so the user must take care to avoid exceeding the maximum frequencies described in the clock generators section. This variation can be mitigated in various ways (see [Section 2.15.2.1.1](#)) if the user wants to continue running from the ROSC at a frequency close to the maximum. Alternatively, the user can use an external clock or the XOSC to provide a stable reference clock and use the PLLs to generate higher frequencies. This will require external components, which will cost board area and increase power consumption.

If an external clock or the XOSC is used then the ROSC can be stopped to save power. However, the reference clock generator and the system clock generator must be switched to an alternate source before doing so.

The ROSC is not affected by SLEEP mode. If required the frequency can be reduced before entering SLEEP mode to save power. On entering DORMANT mode the ROSC is automatically stopped and is restarted in the same configuration when exiting DORMANT mode. If the ROSC is driving clocks at close to their maximum frequencies then it is recommended to drop the frequency before entering SLEEP or DORMANT mode to allow for frequency variation due to changes in environmental conditions during SLEEP or DORMANT mode.

If the user wants to use the ROSC clock externally then it can be output to a GPIO pin using one of the clk_gpcclk0-3 generators.

The following sections describe techniques for mitigating PVT variation of the ROSC frequency. They also provide some interesting design challenges for use in teaching both the effects of PVT and writing software to control real time functions.

NOTE

The ROSC frequency varies with PVT so the user can send its output to the frequency counter and use it to measure any 1 of these 3 variables if the other 2 are known.

2.15.2.1.1. Mitigating ROSC frequency variation due to process

Process varies for two reasons. Firstly, chips leave the factory with a spread of process parameters which cause variation in the ROSC frequency across chips. Secondly, process parameters vary slightly as the chip ages, though this will only be observable over many thousands of hours of operation. To mitigate for process variation, the user can characterise individual chips and program the ROSC frequency accordingly. This is an adequate solution for small numbers of chips but is not suitable for volume production. In such applications the user should consider using the automatic mitigation techniques described below.

2.15.2.1.2. Mitigating ROSC frequency variation due to voltage

Supply voltage varies for two reasons. Firstly, the power supply itself may vary, and secondly, there will be varying on-chip IR drop as chip activity varies. If the application has a minimum performance target then the user needs to calibrate for that application and adjust the ROSC frequency to ensure it always exceeds the minimum required.

2.15.2.1.3. Mitigating ROSC frequency variation due to temperature

Temperature varies for two reasons. Firstly, the ambient temperature may vary, and secondly, the chip temperature will vary as chip activity varies due to self-heating. This can be mitigated by stabilising the temperature using a temperature

2.15.2.1. 环形振荡器

片上环振荡器（第2.17节）无须外部元件。该振荡器在上电时自动启动，用于芯片初始启动阶段的时钟信号。启动频率通常为6MHz，但会随PVT（工艺、供电电压和温度）而变化。频率大致在4至8MHz范围内，且保证处于1.8至12MHz范围内。

对于频率精度要求不高的低成本应用，芯片可继续由ROSC供时。如需更高性能，可按照第2.17节所述通过编程寄存器提升频率。频率会随PVT（工艺、供电电压和温度）波动，用户须谨慎避免超出时钟发生器部分规定的最大频率。若用户希望以接近最大频率继续由ROSC运行，可通过多种方法减缓此类波动（详见第2.15.2.1.1节）。用户也可选择使用外部时钟或XOSC提供稳定的参考时钟，并通过PLL生成更高频率。这将需要额外的外部元件，导致电路板面积增加及功耗上升。

若使用外部时钟或XOSC，则可停止ROSC以节约功耗。但必须先将参考时钟发生器和系统时钟发生器切换至备用时钟源。

ROSC不受SLEEP模式影响。如有必要，可在进入SLEEP模式之前降低频率以节约功耗。进入休眠模式时，ROSC会自动停止；退出休眠模式后，ROSC将以相同配置重新启动。若ROSC驱动的时钟频率接近最大值，建议在进入SLEEP或休眠模式之前降低频率，以适应环境变化导致的频率漂移。

若用户需要将ROSC时钟输出至外部，可使用clk_gpclk0-3发生器之一，将时钟输出到GPIO引脚。

以下章节阐述了缓解ROSC频率PVT变化的技术方案。这些方案同样为教学PVT效应及编写控制实时功能软件提供了有价值的设计挑战。

i 注意

ROSC频率随PVT变化，用户可将其输出接入频率计，在已知另外两个变量条件下，测量三者中的任一变量。

2.15.2.1.1. 缓解因工艺导致的ROSC频率波动

工艺参数变化主要源于两个方面。其一，芯片出厂时工艺参数存在差异，导致芯片间ROSC频率存在变化。其二，芯片随使用时间增长，工艺参数亦会略有变化，但该变化仅在数千小时运行后才可观察到。为缓解工艺变化，用户可针对单颗芯片进行特性测定，并据此编程调整ROSC频率。该方案适用于少量芯片，但不适合大规模批量生产。在此类应用中，用户应考虑采用下述自动缓解技术。

2.15.2.1.2. 缓解电压引起的ROSC频率变化

电源电压的变化有两个原因。首先，电源本身可能波动；其次，随着芯片活动的变化，芯片内部的IR压降也会变化。若应用具有最低性能目标，用户需针对该应用进行校准，并调整ROSC频率，以确保其始终高于最低要求。

2.15.2.1.3. 缓解温度引起的ROSC频率变化

温度变化有两个原因。首先，环境温度可能变化；其次，芯片温度会因自发热随着芯片活动而变化。此问题可通过温度控制环境以及被动或主动冷却措施来加以缓解。

controlled environment and passive or active cooling. Alternatively the user can track the temperature using the on-chip temperature sensor and adjust the ROSC frequency so it remains within the required bounds.

2.15.2.1.4. Automatic mitigation of ROSC frequency variation due to PVT

Techniques for automatic ROSC frequency control avoid the need to calibrate individual chips but require periodic access to a clock reference or to a time reference. If a clock reference is available then it can be used to periodically measure the ROSC frequency and adjust it accordingly. The reference could be the on-chip XOSC which can be turned on periodically for this purpose. This may be useful in a very low power application where it is too costly to run the XOSC continuously and too costly to use the PLLs to achieve high frequencies. If a time reference is available then the user could clock the on-chip RTC from the ROSC and periodically compare it against the time reference, then adjust the ROSC frequency as necessary. Using these techniques the ROSC frequency will drift due to VT variation so the user must take care that these variations do not allow the ROSC frequency to drift out of the acceptable range.

2.15.2.1.5. Automatic overclocking using the ROSC

The datasheet maximum frequencies for any digital device are quoted for worst case PVT. Most chips in most normal environments can run significantly faster than the quoted maximum and can therefore be overclocked. If the RP2040 is running from the ROSC then both the ROSC and the digital components are similarly affected by PVT, so, as the ROSC gets faster, the processors can also run faster. This means the user can overclock from the ROSC then rely on the ROSC frequency tracking with PVT variations. The tracking of ROSC frequency and the processor capability is not perfect and currently there is insufficient data to specify a safe ROSC setting for this mode of operation, so some experimentation is required.

This mode of operation will maximise processor performance but will lead to variations in the time taken to complete a task, which may be unacceptable in some applications. Also, if the user wants to use frequency sensitive interfaces such as USB or UART then the XOSC and PLL must be used to provide a precise clock for those components.

2.15.2.2. Crystal Oscillator

The Crystal Oscillator ([Section 2.16](#)) provides a precise, stable clock reference and should be used where accurate timing is required and no suitable external clocks are available. The frequency is determined by the external crystal and the oscillator supports frequencies in the range 1MHz to 15MHz. The on-chip PLLs can be used to synthesise higher frequencies if required. The RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#)) uses a 12MHz crystal. Using the XOSC and the PLLs, the on-chip components can be run at their maximum frequencies. Appropriate margin is built into the design to tolerate up to 1000ppm variation in the XOSC frequency.

The XOSC is inactive on power up. If required it must be enabled in software. XOSC startup takes several milliseconds and the software must wait for the XOSC_STABLE flag to be set before starting the PLLs and before changing any clock generators to use it. Prior to that the output from the XOSC may be non-existent or may have very short pulse widths which will corrupt logic if used. Once it is running the reference clock (clk_ref) and the system clock (clk_sys) can be switched to run from the XOSC and the ROSC can be stopped to save power.

The XOSC is not affected by SLEEP mode. It is automatically stopped and restarted in the same configuration when entering and exiting DORMANT mode.

If the user wants to use the XOSC clock externally then it can be output to a GPIO pin using one of the clk_gpclk0-3 generators. It cannot be taken directly from the XIN or XOUT pins.

2.15.2.3. External Clocks

If external clocks exist in your hardware design then they can be used to clock the RP2040 either on their own or in conjunction with the XOSC or ROSC. This will potentially save power and will allow components on the RP2040 to be run synchronously with external components to simplify data transfer between chips. External clocks can be input on the

控制环境和冷却。或者，用户可利用片上温度传感器监测温度，并调整ROSC频率，使其保持在所需范围内。

2.15.2.1.4 由于PVT引起的ROSC频率变化的自动缓解

自动ROSC频率控制技术避免了单独校准芯片的需求，但需要周期性地访问时钟参考或时间参考。若时钟参考可用，可周期性地测量ROSC频率并据此调整。该参考信号可为片上XOSC，且可为此目的周期性开启。此方法在功耗极低的应用中尤其有用，当连续运行XOSC代价过高，且使用PLL实现高频同样代价昂贵时。若时间参考可用，用户可用ROSC为片上RTC提供时钟，周期性将其与时间参考比较，并据此调整ROSC频率。应用此技术时，ROSC频率会因VT变化而漂移，用户必须确保这种变化不会使ROSC频率超出可接受范围。

2.15.2.1.5 使用ROSC进行自动超频

任何数字器件的数据手册中所列最大频率均基于最恶劣的PVT条件。大多数芯片在正常环境下的实际运行速度通常显著高于此最大值，因此可进行超频。若RP2040由ROSC驱动，则ROSC及数字组件均受PVT影响，随着ROSC频率提升，处理器同样得以加速。这意味着用户可通过ROSC超频，并依赖ROSC频率随PVT变化进行跟踪。ROSC频率与处理器性能的跟踪尚不完善，目前缺乏足够数据以明确该运行模式下的安全ROSC设置，因而需进行一定实验。

该运行模式虽可最大化处理器性能，但任务完成时间将出现波动，可能不适用于部分应用场景。此外，若用户需使用对频率敏感的接口（如USB或UART），则必须采用XOSC及PLL为这些组件提供精确时钟。

2.15.2.2. 晶体振荡器

晶体振荡器（第2.16节）提供精确且稳定的时钟参考，应在需要准确计时且无合适外部时钟时使用。频率由外部晶体决定，振荡器支持1MHz至15MHz范围内的频率。片上相位锁定环（PLL）可用于合成更高频率（如有需要）。RP2040参考设计（参见《使用RP2040的硬件设计》中的最小设计示例）使用12MHz晶体。利用XOSC和PLL，片上组件能够以最高频率运行。设计中预留了足够裕量，能够容忍XOSC频率高达1000ppm的偏差。

XOSC在上电时处于非激活状态，如需使用必须通过软件启用。XOSC启动需数毫秒，软件必须等待XOSC_STABLE标志被置位后，方可启动PLL并切换任何时钟生成器至该源。在此之前，XOSC输出可能存在极短脉冲，若被使用将导致逻辑损坏。一旦运行，参考时钟（clk_ref）和系统时钟（clk_sys）可以切换至由XOSC驱动，同时ROSC可被停止以节省功耗。

XOSC不受休眠模式影响。在进入和退出休眠模式时，XOSC会以相同配置自动停止并重新启动。

若用户需将XOSC时钟输出至外部，可通过clk_gpcclk0-3生成器之一，将其输出至GPIO引脚。时钟信号不可直接从XIN或XOUT引脚获取。

2.15.2.3. 外部时钟

若您的硬件设计包含外部时钟，亦可单独或结合XOSC或ROSC，为RP2040供时。此举不仅可节省功耗，还能使RP2040组件与外部组件同步运行，简化芯片间数据传输。外部时钟可输入至

GPIN0 & GPIN1 GPIO inputs and on the XIN input to the XOSC. If the XIN input is used in this way the XOSC must be configured to pass through the XIN signal. All 3 inputs are limited to 50MHz but the on-chip PLLs can be used to synthesise higher frequencies from the XIN input if required. If the frequency accuracy of the external clocks is poorer than 1000ppm then the generated clocks should not be run at their maximum frequencies because they may exceed their design margins.

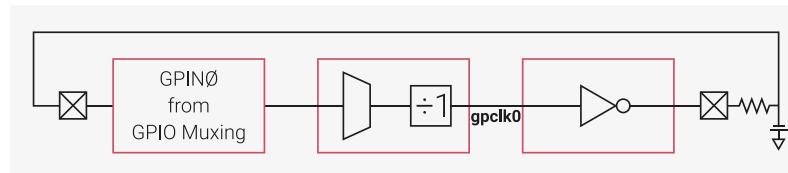
Once the external clocks are running, the reference clock (clk_ref) and the system clock (clk_sys) can be switched to run from the external clocks and the ROSC can be stopped to save power.

The external clock sources are not affected by SLEEP mode or DORMANT mode.

2.15.2.4. Relaxation Oscillators

If the user wants to use external clocks to replace or supplement the other clock sources but does not have an appropriate clock available, then 1 or 2 relaxation oscillators can be constructed using external passive components. Simply send the clock source (GPIN0 or GPIN1) to one of the gpcclk0-3 generators, invert it through the GPIO inverter OUTOVER and connect back to the clock source input via an RC circuit.

Figure 29. Simple relaxation oscillator example



The frequency of clocks generated from relaxation oscillators will depend on the delay through the chip and the drive current from the GPIO output both of which vary with PVT. They will also depend on the quality and accuracy of the external components. It may be possible to improve the frequency accuracy using more elaborate external components such as ceramic resonators but that will increase cost and complexity and can never rival the XOSC. For that reason they are not discussed here. Given that these oscillators will not achieve 1000ppm then they cannot be used to drive internal clocks at their maximum frequencies.

The relaxation oscillators are not affected by SLEEP mode or DORMANT mode.

2.15.2.5. PLLs

The PLLs ([Section 2.18](#)) are used to provide fast clocks when running from the XOSC (or an external clock source driven into the XIN pin). In a fully featured application the USB PLL provides a fixed 48MHz clock to the ADC and USB while clk_rtc and clk_ref are driven from the XOSC or external source. This allows the user to drive clk_sys from the system PLL and vary the frequency according to demand to save power without having to change the setups of the other clocks. clk_peri can be driven either from the fixed frequency USB PLL or from the variable frequency system PLL. If clk_sys never needs to exceed 48MHz then one PLL can be used and the divider in the clk_sys clock generator can be used to scale the clk_sys frequency according to demand.

When a PLL is started, its output cannot be used until the PLL locks as indicated by the LOCK bit in the STATUS register. Thereafter the PLL output cannot be used during changes to the reference clock divider, the output dividers or the bypass mode. The output can be used during feedback divisor changes with the proviso that the output frequency may overshoot or undershoot on large changes to the feedback divisor. For more information, see [Section 2.18](#).

If the PLL reference clock is accurate to 1000ppm then the PLLs can be used to drive clocks at their maximum frequency because the frequency of the generated clocks will be within the margins allowed in the design.

The PLLs are not affected by SLEEP mode. If the user wants to save power in SLEEP mode then all clock generators must be switched away from the PLLs and they must be stopped in software before entering SLEEP mode. The PLLs are not stopped and restarted automatically when entering and exiting DORMANT mode. If they are left running on entry to DORMANT mode they will be corrupted and will generate out of control clocks that will consume power unnecessarily. This happens because their reference clock from XOSC will be stopped. It is therefore essential to switch all clock generators away from the PLLs and stop the PLLs in software before entering DORMANT mode.

GPIN0 和 GPIN1 GPIO 输入以及 XOSC 的 XIN 输入。如果以此方式使用 XIN 输入，则必须将 XOSC 配置为传递 XIN 信号。所有三个输入均限制为 50MHz，但片上 PLL 可根据需要利用 XIN 输入合成更高频率。如果外部时钟的频率精度低于 1000ppm，则生成的时钟不应以最大频率运行，以免超出设计容限。

外部时钟开始运行后，参考时钟（clk_ref）和系统时钟（clk_sys）可以切换至外部时钟驱动，同时停止 ROSC 以节省功耗。

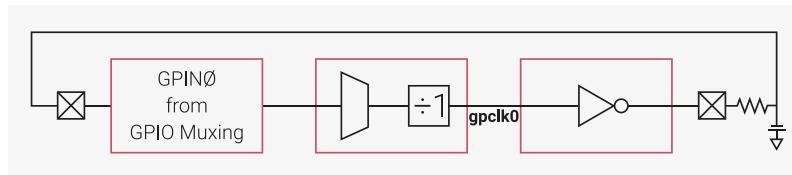
外部时钟源不受 SLEEP 模式或休眠模式影响。

2.15.2.4. 松弛振荡器

若用户希望使用外部时钟替代或补充其他时钟源，但无合适时钟可用，则可使用外部无源元件构建一至两个松弛振荡器。

简单地将时钟源（GPIN0 或 GPIN1）发送至 gpclk0-3 生成器之一，并通过 GPIO 反相器进行反相 OUTOVER，再通过 RC 电路连接回时钟源输入。

图29。简单
松弛振荡器
示例



由松弛振荡器产生的时钟频率取决于芯片内部的延迟及 GPIO 输出的驱动电流，二者均随 PVT 变化。此外，还取决于外部元件的质量与精度。虽可通过使用如陶瓷谐振器等更复杂的外部元件来提升频率精度，但这将增加成本和复杂度，且始终无法媲美 XOSC。因此，此处不予讨论。鉴于这些振荡器无法达到 1000ppm 的精度，故不可用于驱动内部时钟达到最大频率。

松弛振荡器不受 SLEEP 模式及休眠模式影响。

2.15.2.5. PLLs

PLL（参见第2.18节）用于在从XOSC（或驱动至XIN引脚的外部时钟源）运行时提供高速时钟。在功能完备的应用中，USB PLL为ADC和USB提供固定的48MHz时钟，而clk_rtc和clk_ref由XOSC或外部时钟源驱动。这允许用户通过系统PLL驱动clk_sys，并根据需求调节频率以节能，而无需更改其他时钟的设置。clk_peri可以由固定频率的USB PLL或可变频率的系统PLL驱动。如果clk_sys频率不超过48MHz，则可仅使用一个PLL，并通过clk_sys时钟生成器中的分频器根据需求调整clk_sys频率。

PLL启动后，在PLL锁定并由STATUS寄存器的LOCK位指示之前，其输出不可用。

此后，在更改参考时钟分频器、输出分频器或旁路模式时，PLL输出不可用。在反馈除数改变期间，输出可被使用，但在反馈除数发生较大变化时，输出频率可能会出现超过或低于设定值的情况。详见第2.18节。

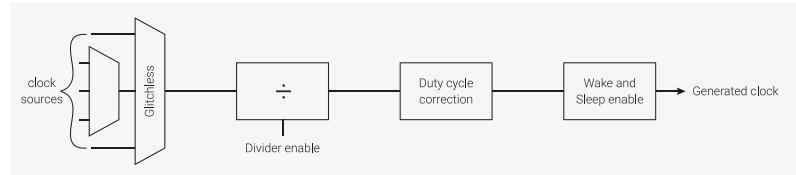
若PLL参考时钟精度达到1000ppm，则PLL可用于以最大频率驱动时钟，因为生成时钟的频率将处于设计允许的范围内。

PLL不受休眠模式影响。若用户欲在休眠模式下节能，必须将所有时钟发生器切换离开PLL，且在进入休眠模式前通过软件停止它们。进入及退出休眠模式时，PLL不会自动停止或重启。若在进入休眠模式时PLL仍运行，PLL将被破坏，产生失控时钟，导致不必要的功耗。这是因其参考时钟XOSC会被停止。因此，进入休眠模式前，务必停止所有时钟发生器并离开PLL，并通过软件停止PLL。

2.15.3. Clock Generators

The clock generators are built on a standard design which incorporates clock source multiplexing, division, duty cycle correction and SLEEP mode enabling. To save chip area and power, the individual clock generators do not support all features.

Figure 30. A generic clock generator



2.15.3.1. Instances

RP2040 has several clock generators which are listed below.

Table 205. RP2040 clock generators

Clock	Description	Nominal Frequency
<code>clk_gpout0</code>	Clock output to GPIO. Can be used to clock external devices or debug on chip clocks with a logic analyser or oscilloscope.	N/A
<code>clk_gpout1</code>		
<code>clk_gpout2</code>		
<code>clk_gpout3</code>		
<code>clk_ref</code>	Reference clock that is always running unless in DORMANT mode. Runs from Ring Oscillator (ROSC) at power-up but can be switched to Crystal Oscillator (XOSC) for more accuracy.	6 - 12MHz
<code>clk_sys</code>	System clock that is always running unless in DORMANT mode. Runs from <code>clk_ref</code> at power-up but is typically switched to a PLL.	125MHz
<code>clk_peri</code>	Peripheral clock. Typically runs from <code>clk_sys</code> but allows peripherals to run at a consistent speed if <code>clk_sys</code> is changed by software.	12 - 125MHz
<code>clk_usb</code>	USB reference clock. Must be 48MHz.	48MHz
<code>clk_adc</code>	ADC reference clock. Must be 48MHz.	48MHz
<code>clk_rtc</code>	RTC reference clock. The RTC divides this clock to generate a 1 second reference.	46875Hz

NOTE

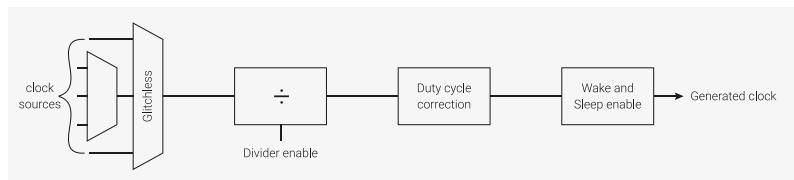
`clk_sys` (and `clk_peri`) have a maximum frequency of 133MHz across all process, voltage and temperature variations. You can achieve 200MHz by running at an elevated core supply (DVDD) and setting `VREG_VSEL` to 1.15V. For more information, see [Section 2.10.6](#).

For a full list of clock sources for each clock generator see the appropriate `CTRL` register. For example, `CLK_SYS_CTRL`.

2.15.3. 时钟发生器

时钟发生器采用标准设计，集成时钟源多路复用、分频、占空比校正及休眠模式启用功能。为节省芯片面积和功耗，单个时钟发生器并不支持所有功能。

图30。通用时钟发生器



2.15.3.1. 实例

RP2040 配备多个时钟发生器，列举如下。

表205. RP2040
时钟发生器

时钟	描述	额定频率
<code>clk_gpout0</code>	时钟信号输出至 GPIO，可用于驱动外部设备，或使用逻辑分析仪及示波器调试片上时钟。	不适用
<code>clk_gpout1</code>		
<code>clk_gpout2</code>		
<code>clk_gpout3</code>		
<code>clk_ref</code>	参考时钟，除休眠模式外始终运行。上电时由环形振荡器（ROSC）驱动，可切换至晶体振荡器（XOSC）以提升精度。	6 - 12MHz
<code>clk_sys</code>	系统时钟，除休眠模式外始终运行。上电时由 <code>clk_ref</code> 驱动，通常切换至 PLL。	125MHz
<code>clk_peri</code>	外设时钟。通常由 <code>clk_sys</code> 运行，但若软件更改 <code>clk_sys</code> ，则允许外围设备以恒定速度运行。	12 - 125MHz
<code>clk_usb</code>	USB参考时钟，必须为48MHz。	48MHz
<code>clk_adc</code>	ADC参考时钟，必须为48MHz。	48MHz
<code>clk_rtc</code>	RTC参考时钟。RTC通过分频该时钟以产生1秒参考信号。	46875Hz

① 注意

`clk_sys`（及 `clk_peri`）在所有工艺、电压及温度变化条件下的最高频率为133MHz。
通过提升核心电源（DVDD）并将VREG VSEL设置为1.15V，可实现200MHz频率。详见第2.10.6节。

有关每个时钟发生器之完整时钟源列表，请参阅相应的 `CTRL` 寄存器。例如，`CLK_SYS_CTRL`。

2.15.3.2. Multiplexers

All clock generators have a multiplexer referred to as the auxiliary (aux) mux. This mux has a conventional design whose output will glitch when changing the select control. Two clock generators (`clk_sys` and `clk_ref`) have an additional multiplexer, referred to as the glitchless mux. The glitchless mux can switch between clock sources without generating a glitch on the output.

Clock glitches should be avoided at all costs because they may corrupt the logic running on that clock. This means that any clock generator with only an aux mux must be disabled while switching the clock source. If the clock generator has a glitchless mux (`clk_sys` and `clk_ref`), then the glitchless mux should switch away from the aux mux while changing the aux mux source. The clock generators require 2 cycles of the source clock to stop the output and 2 cycles of the new source to restart the output. The user must wait for the generator to stop before changing the auxiliary mux, and therefore must be aware of the source clock frequency.

The glitchless mux is only implemented for always-on clocks. On RP2040 the always-on clocks are the reference clock (`clk_ref`) and the system clock (`clk_sys`). Such clocks must run continuously unless the chip is in DORMANT mode. The glitchless mux has a status output (`SELECTED`) which indicates which source is selected and can be read from software to confirm that a change of clock source has been completed.

The recommended control sequences are as follows.

To switch the glitchless mux:

- switch the glitchless mux to an alternate source
- poll the `SELECTED` register until the switch is completed

To switch the auxiliary mux when the generator has a glitchless mux:

- switch the glitchless mux to a source that isn't the aux mux
- poll the `SELECTED` register until the switch is completed
- change the auxiliary mux select control
- switch the glitchless mux back to the aux mux
- if required, poll the `SELECTED` register until the switch is completed

To switch the auxiliary mux when the generator does not have a glitchless mux:

- disable the clock divider
- wait for the generated clock to stop (2 cycles of the clock source)
- change the auxiliary mux select control
- enable the clock divider
- if required, wait for the clock generator to restart (2 cycles of the clock source)

See [Section 2.15.6.1](#) for a code example of this.

2.15.3.3. Divider

A fully featured divider divides by 1 or a fractional number in the range 2.0 to $2^{24}-0.01$. Fractional division is achieved by toggling between 2 integer divisors therefore it yields a jittery clock which may not be suitable for some applications. For example, when dividing by 2.4 the divider will divide by 2 for 3 cycles and by 3 for 2 cycles. For divisors with large integer components the jitter will be much smaller and less critical.

2.15.3.2. 多路复用器

所有时钟发生器均设有称为辅助（aux）多路复用器的复用器，该多路复用器采用传统设计，切换选择控制时输出会产生毛刺。两个时钟发生器（`clk_sys`和`clk_ref`）设有一个额外的多路复用器，称为无毛刺多路复用器。该无毛刺多路复用器能够在切换时钟源时，避免在输出端产生毛刺。

必须全力避免时钟毛刺，因为其可能损坏基于该时钟运行的逻辑电路。这意味着，任何仅配备辅助多路复用器的时钟发生器，在切换时钟源时必须被禁用。若时钟发生器配备无毛刺多路复用器（`clk_sys`和`clk_ref`），则在更改辅助多路复用器的信号源时，应先通过无毛刺多路复用器切换离开辅助多路复用器。时钟发生器需耗费源时钟两个周期以停止输出，且需两个新源时钟周期以重新启动输出。用户必须等待发生器完全停止后方可更改辅助多路复用器，故需准确掌握源时钟频率。

无毛刺多路复用器仅适用于始终开启的时钟信号。在RP2040芯片中，始终开启的时钟包括参考时钟（`clk_ref`）及系统时钟（`clk_sys`）。此类时钟必须持续运行，除非芯片处于休眠模式。无误差复用器具有状态输出（SELECTED），用于指示所选源，且可通过软件读取以确认时钟源切换已完成。

推荐的控制序列如下：

切换无误差复用器：

- 将无误差复用器切换至备用源
- 轮询SELECTED寄存器，直至切换完成

当发生器具备无误差复用器时，切换辅助复用器：

- 将无误差复用器切换至非辅助复用器的源
- 轮询SELECTED寄存器，直至切换完成
- 更改辅助复用器选择控制
- 将无误差复用器切換回辅助复用器
- 如有需要，轮询SELECTED寄存器，直至切换完成

当发生器不具备无误差复用器时，切换辅助复用器：

- 禁用时钟分频器
- 等待生成的时钟停止（时钟源的两个周期）
- 更改辅助复用器选择控制
- 启用时钟分频器
- 如有需要，等待时钟发生器重启（时钟源的两个周期）

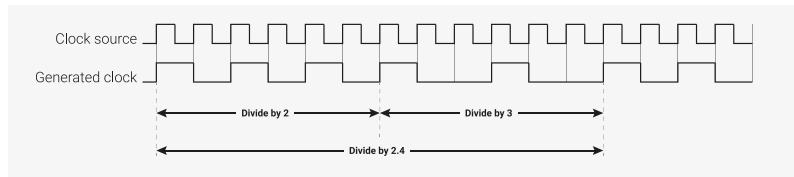
详见第2.15.6.1节代码示例。

2.15.3.3. 分频器

一个功能齐全的分频器可将频率除以1或介于2.0至 $2^{24}-0.01$ 之间的分数。分数除法通过在两个整数除数之间切换实现，因此输出时钟可能会产生抖动，某些应用可能不适用。

例如，当除数为2.4时，分频器将连续3个周期除以2，随后连续2个周期除以3。对于具有较大整数成分的除数，抖动显著减小且影响较小。

Figure 31. An example of fractional division.



All dividers support on-the-fly divisor changes meaning the output clock will switch cleanly from one divisor to another. The clock generator does not need to be stopped during clock divisor changes. It does this by synchronising the divisor change to the end of the clock cycle. Similarly, the enable is synchronised to the end of the clock cycle so will not generate glitches when the clock generator is enabled or disabled. Clock generators for always-on clocks are permanently enabled and therefore do not have an enable control.

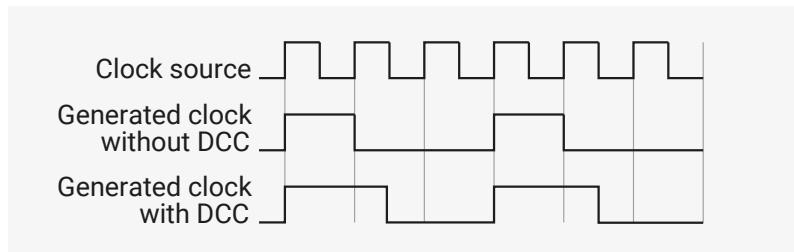
In the event that a clock generator locks up and never completes the current clock cycle it can be forced to stop using the KILL control. This may result in an output glitch which may corrupt the logic driven by the clock. It is therefore recommended the destination logic is reset prior to this operation. It is worth mentioning that this clock generator design has been used in numerous chips and has never been known to lock up. The KILL control is inelegant and unnecessary and should not be used as an alternative to the enable. Clock generators for always-on clocks are permanently active and therefore do not have a KILL control.

2.15.3.4. Duty Cycle Correction

The divider operates on the rising edge of the input clock and so does not generate an even duty cycle clock when dividing by odd numbers.

Divide by 3 will give a duty cycle of 33.3%, divide by 5 will be 40% etc. If enabled, the duty cycle correction logic will shift the falling edge of the output clock to the falling edge of the input clock and restore a 50% duty cycle. The duty cycle correction can be enabled and disabled while the clock is running. It will not operate when dividing by an even number.

Figure 32. An example of duty_cycle_correction.



2.15.3.5. Clock Enables

Each clock goes to multiple destinations and, with a few exceptions, there are 2 enables for each destination. The `WAKE_EN` registers are used to enable the clocks when the system is awake and the `SLEEP_EN` registers are used to enable the clocks when the system is in sleep mode. The purpose of these enables is to reduce power in the clock distribution networks for components that are not being used. It is worth noting that a component which is not clocked will retain its configuration so can be restarted quickly.

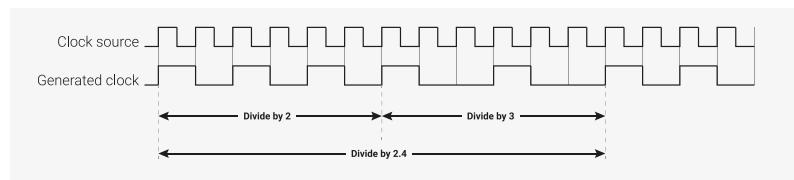
NOTE

The `WAKE_EN` and `SLEEP_EN` registers reset to `0x1`, which means that by default all clocks are enabled. The programmer only needs to use this feature if they desire a low-power design.

2.15.3.5.1. Clock Enable Exceptions

The processor cores do not have clock enables because they require a clock at all times to manage their own power saving features.

图31。分数除法的示例。



所有分频器均支持动态变更除数，输出时钟可平滑切换至新除数。

时钟发生器在改变时钟除数时无需停止。此功能通过将除数变更同步至时钟周期末实现。同样，启用信号与时钟周期结束同步，因此在时钟生成器启用或禁用时不会产生毛刺。用于始终开启的时钟生成器永久启用，因此不存在启用控制。

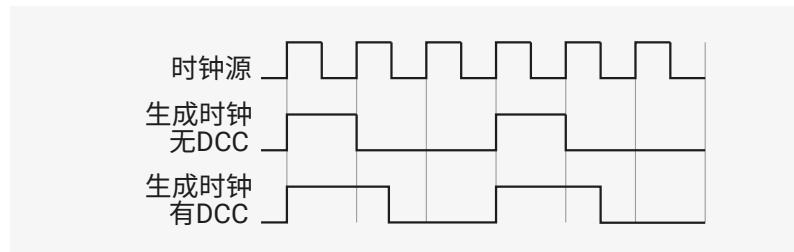
如果时钟生成器锁死，且当前时钟周期永远无法完成，可通过 KILL 控制强制停止。此操作可能导致输出毛刺，进而破坏由该时钟驱动的逻辑。因此，建议在执行此操作前先复位目标逻辑。值得一提的是，该时钟生成器设计已应用于众多芯片，且从未出现锁死情况。KILL 控制方法不够优雅且无必要，切勿用作启用控制的替代方案。始终开启的时钟生成器永久激活，因此不设 KILL 控制。

2.15.3.4. 占空比校正

分频器于输入时钟上升沿工作，故除数为奇数时无法生成均匀占空比的时钟信号。

除以3时占空比为33.3%，除以5时为40%，依此类推。如启用，占空比校正逻辑将把输出时钟的下降沿调整至输入时钟的下降沿，从而恢复50%的占空比。占空比校正功能可在时钟运行时启用或禁用。除以偶数时，该功能无效。

图32。duty_cycle_correction 的示例。



2.15.3.5 时钟使能

每个时钟输出连接至多个目标，除少数例外，每个目标具备两个使能信号。`WAKE_EN`寄存器用于系统唤醒状态下的时钟使能，`SLEEP_EN`寄存器用于系统睡眠状态下的时钟使能。设置这些使能旨在降低未使用组件时钟分布网络的功耗。值得注意的是，未被时钟驱动的组件将保留其配置，因此能够快速重启。

注意

寄存器 `WAKE_EN` 和 `SLEEP_EN` 复位为 `0x1`，意味着默认情况下所有时钟均处于启用状态。程序员仅在设计低功耗方案时需要使用此功能。

2.15.3.5.1 时钟启用例外

处理器内核无时钟启用选项，因其始终需要时钟以管理自身节能特性。

`clk_sys_busfabric` cannot be disabled in wake mode because that would prevent the cores from accessing any chip registers, including those that control the clock enables.

`clk_sys_clocks` does not have a wake mode enable because disabling it would prevent the cores from accessing the clocks control registers.

The gpclks do not have clock enables.

2.15.3.5.2. System Sleep Mode

System sleep mode is entered automatically when both cores are in sleep and the DMA has no outstanding transactions. In system sleep mode, the clock enables described in the previous paragraphs are switched from the `WAKE_EN` registers to the `SLEEP_EN` registers. The intention is to reduce power consumed in the clock distribution networks when the chip is inactive. If the user has not configured the `WAKE_EN` and `SLEEP_EN` registers then system sleep will do nothing.

There is little value in using system sleep without taking other measures to reduce power before the cores are put to sleep. Things to consider include:

- stop unused clock sources such as the PLLs and Crystal Oscillator
- reduce the frequencies of generated clocks by increasing the clock divisors
- stop external clocks

For maximum power saving when the chip is inactive, the user should consider DORMANT (see [Section 2.11.3](#)) mode in which clocks are sourced from the Crystal Oscillator and/or the Ring Oscillator and those clock sources are stopped.

2.15.4. Frequency Counter

The frequency counter measures the frequency of internal and external clocks by counting the clock edges seen over a test interval. The interval is defined by counting cycles of `clk_ref` which must be driven either from XOSC or from a stable external source of known frequency.

The user can pick between accuracy and test time using the `FC0_INTERVAL` register. [Table 206](#) shows the trade off.

Table 206. Frequency Counter Test Interval vs Accuracy

Interval Register	Test Interval	Accuracy
0	1µs	2048kHz
1	2µs	1024kHz
2	4µs	512kHz
3	8µs	256kHz
4	16µs	128kHz
5	32µs	64kHz
6	64µs	32kHz
7	125µs	16kHz
8	250µs	8kHz
9	500µs	4kHz
10	1ms	2kHz
11	2ms	1kHz
12	4ms	500Hz
13	8ms	250Hz

`clk_sys_busfabric`不可在唤醒模式下禁用，否则将阻止内核访问包括时钟启用控制在内的任何芯片寄存器。

`clk_sys_clocks`无唤醒模式启用选项，禁用该时钟将阻止内核访问时钟控制寄存器。

gpclks无时钟启用功能。

2.15.3.5.2 系统睡眠模式

当两个内核均处于睡眠状态且DMA无未完成事务时，系统将自动进入睡眠模式。在系统睡眠模式下，前述时钟使能从`WAKE_EN`寄存器切换至`SLEEP_EN`寄存器。其目的是在芯片处于非活动状态时减少时钟分配网络的功耗。如果用户未配置`WAKE_EN`和`SLEEP_EN`寄存器，系统睡眠将不会产生任何效果。

在内核进入睡眠之前，如未采取其他降低功耗的措施，单独使用系统睡眠意义有限。需考虑的事项包括：

- 停止未使用的时钟源，例如PLL和晶体振荡器
- 通过增加时钟分频器降低生成时钟的频率
- 停止外部时钟

为实现芯片非活动状态下的最大功耗节省，用户应考虑使用DORMANT模式（详见第2.11.3节），该模式下时钟源为晶体振荡器和/或环形振荡器，并停止这些时钟源。

2.15.4. 频率计数器

频率计数器通过在测试间隔内计数时钟的边沿数来测量内外部时钟频率。测试间隔由`clk_ref`的周期数定义，`clk_ref`必须由XOSC或已知频率的稳定外部源驱动。

用户可以通过`FC0_INTERVAL`寄存器在准确度与测试时间之间进行选择。表206展示了该折衷关系。

表206。频率计测试间隔与准确度

间隔寄存器	测试间隔	准确度
0	1μ秒	2048kHz
1	2μ秒	1024kHz
2	4μ秒	512kHz
3	8μ秒	256kHz
4	16μ秒	128kHz
5	32μ秒	64kHz
6	64μ秒	32kHz
7	125μ秒	16kHz
8	250μ秒	8kHz
9	500μ秒	4kHz
10	1ms	2kHz
11	2ms	1kHz
12	4ms	500Hz
13	8ms	250Hz

Interval Register	Test Interval	Accuracy
14	16ms	125Hz
15	32ms	62.5Hz

2.15.5. Resus

It is possible to write software that inadvertently stops `clk_sys`. This will normally cause an unrecoverable lock-up of the cores and the on-chip debugger, leaving the user unable to trace the problem. To mitigate against that, an automatic resuscitation circuit is provided which will switch `clk_sys` to a known good clock source if no edges are detected over a user-defined interval. The known good source is `clk_ref` which can be driven from the XOSC, ROSC or an external source.

The resus block counts edges on `clk_sys` during a timeout interval controlled by `clk_ref`, and forces `clk_sys` to be driven from `clk_ref` if no `clk_sys` edges are detected. The interval is programmable via `CLK_SYS_RESUS_CTRL`.

⚠️ WARNING

There is no way for resus to revive the chip if `clk_ref` is also stopped.

To enable the resus, the programmer must set the timeout interval and then set the `ENABLE` bit in `CLK_SYS_RESUS_CTRL`. To detect a resus event, the `CLK_SYS_RESUS` interrupt must be enabled by setting the interrupt enable bit in `INTE`. The `CLOCKS_DEFAULT_IRQ` (see [Section 2.3.2](#)) must also be enabled at the processor.

Resus is intended as a debugging aid. The intention is for the user to trace the software error that triggered the resus, then correct the error and reboot. It is possible to continue running after a resus event by reconfiguring `clk_sys` then clearing the resus by writing the `CLEAR` bit in `CLK_SYS_RESUS_CTRL`. However, it should be noted that a resus can be triggered by `clk_sys` running more slowly than expected and that could result in a `clk_sys` glitch when resus is triggered. That glitch could corrupt the chip. This would be a rare event but is tolerable in a debugging scenario. However it is unacceptable in normal operation therefore it is recommended to only use resus for debug.

⚠️ WARNING

Resus is a debugging aid and should not be used as a means of switching clocks in normal operation.

2.15.6. Programmer's Model

2.15.6.1. Configuring a clock generator

The SDK defines an enum of clocks:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h Lines 30 - 42

```

30 typedef enum clock_num_rp2040 {
31     clk_gpout0 = 0, ///< Select CLK_GPOUT0 as clock source
32     clk_gpout1 = 1, ///< Select CLK_GPOUT1 as clock source
33     clk_gpout2 = 2, ///< Select CLK_GPOUT2 as clock source
34     clk_gpout3 = 3, ///< Select CLK_GPOUT3 as clock source
35     clk_ref = 4, ///< Select CLK_REF as clock source
36     clk_sys = 5, ///< Select CLK_SYS as clock source
37     clk_peri = 6, ///< Select CLK_PERI as clock source
38     clk_usb = 7, ///< Select CLK_USB as clock source
39     clk_adc = 8, ///< Select CLK_ADC as clock source
40     clk_rtc = 9, ///< Select CLK_RTC as clock source
41     CLK_COUNT

```

间隔寄存器	测试间隔	准确度
14	16ms	125Hz
15	32ms	62.5Hz

2.15.5. Resus

可能出现软件无意中停止 `clk_sys` 的情况。这通常会导致核心及片上调试器不可恢复的死锁，用户无法追踪问题。为防止此类情况，系统配备了自动复苏电路，如在用户设定的时间间隔内未检测到边沿，将自动将 `clk_sys` 切换至已知的良好时钟源。该良好时钟源为 `clk_ref`，可由 XOSC、ROSC 或外部源驱动。

复苏模块在由 `clk_ref` 控制的超时区间内计数 `clk_sys` 的边沿，若未检测到任何边沿，强制 `clk_sys` 由 `clk_ref` 驱动。此时间间隔可通过 `CLK_SYS_RESUS_CTRL` 寄存器进行编程设置。

● 警告

若 `clk_ref` 亦停止，复苏电路无法恢复芯片功能。

为启用 resus，程序员必须设置超时间隔，然后在 `CLK_SYS_RESUS_CTRL` 中设置 `ENABLE` 位。

为检测 resus 事件，必须通过在 INTE 中设置中断使能位启用 `CLK_SYS_RESUS` 中断。处理器还必须启用 `CLOCKS_DEFAULT_IRQ`（详见第 2.3.2 节）。

Resus 旨在作为调试辅助工具，目的是让用户追踪触发 resus 的软件错误，进而修正该错误并重启系统。通过重新配置 `clk_sys` 后，写入 `CLK_SYS_RESUS_CTRL` 中的 `CLEAR` 位即可清除 resus 并继续运行。但应注意，resus 可能由 `clk_sys` 运行速度低于预期引发，进而导致 resus 触发时出现 `clk_sys` 故障。

该故障可能会损坏芯片。此类事件虽属罕见，但在调试环境中可被接受。然而，该操作在正常运行中不可接受，因此建议仅将 resus 用于调试。

● 警告

Resus 为调试辅助工具，不应用于正常运行中的时钟切换。

2.15.6. 程序员模型

2.15.6.1. 配置时钟发生器

SDK 定义了一个时钟枚举类型：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h 第 30 行至 42 行

```

30 typedef enum clock_num_rp2040 {
31     clk_gpout0 = 0, ///< 选择 CLK_GPOUT0 作为时钟源
32     clk_gpout1 = 1, ///< 选择 CLK_GPOUT1 作为时钟源
33     clk_gpout2 = 2, ///< 选择 CLK_GPOUT2 作为时钟源
34     clk_gpout3 = 3, ///< 选择 CLK_GPOUT3 作为时钟源
35     clk_ref = 4, ///< 选择 CLK_REF 作为时钟源
36     clk_sys = 5, ///< 选择 CLK_SYS 作为时钟源
37     clk_peri = 6, ///< 选择 CLK_PERI 作为时钟源
38     clk_usb = 7, ///< 选择 CLK_USB 作为时钟源
39     clk_adc = 8, ///< 选择 CLK_ADC 作为时钟源
40     clk_rtc = 9, ///< 选择 CLK_RTC 作为时钟源
41     CLK_COUNT

```

```
42 } clock_num_t;
```

And also a struct to describe the registers of a clock generator:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h Lines 100 - 121

```
100 typedef struct {
101     _REG_(CLOCKS_CLK_GPOUT0_CTRL_OFFSET) // CLOCKS_CLK_GPOUT0_CTRL
102     // Clock control, can be changed on-the-fly (except for auxsrc)
103     // 0x00100000 [20]    NUDGE          (0) An edge on this signal shifts the phase of the
104     // output by...
105     // 0x00030000 [17:16] PHASE        (0x0) This delays the enable signal by up to 3 cycles
106     // of the...
107     // 0x00001000 [12]    DC50          (0) Enables duty cycle correction for odd divisors
108     // 0x00000800 [11]    ENABLE         (0) Starts and stops the clock generator cleanly
109     // 0x00000400 [10]    KILL           (0) Asynchronously kills the clock generator
110     // 0x000001e0 [8:5]   AUXSRC        (0x0) Selects the auxiliary clock source, will glitch
111     // when switching
112     io_rw_32 ctrl;
113
114     _REG_(CLOCKS_CLK_GPOUT0_DIV_OFFSET) // CLOCKS_CLK_GPOUT0_DIV
115     // Clock divisor, can be changed on-the-fly
116     // 0xffffffff00 [31:8] INT          (0x000001) Integer component of the divisor, 0 ->
117     // divide by 2^16
118     // 0x000000ff [7:0]   FRAC          (0x00) Fractional component of the divisor
119     io_rw_32 div;
120
121     _REG_(CLOCKS_CLK_GPOUT0_SELECTED_OFFSET) // CLOCKS_CLK_GPOUT0_SELECTED
122     // Indicates which SRC is currently selected by the glitchless mux (one-hot)
123     // 0xffffffff [31:0] CLK_GPOUT0_SELECTED (0x00000001) This slice does not have a
124     // glitchless mux (only the...
125     io_ro_32 selected;
126 } clock_hw_t;
```

To configure a clock, we need to know the following pieces of information:

- The frequency of the clock source
- The mux / aux mux position of the clock source
- The desired output frequency

The SDK provides `clock_configure` to configure a clock:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/p2_common/hardware_clocks/clocks.c Lines 40 - 133

```
40 static void clock_configure_internal(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
41                                     uint32_t actual_freq, uint32_t div) {
42     clock_hw_t *clock_hw = &clocks_hw->clk[clock];
43
44     // If increasing divisor, set divisor before source. Otherwise set source
45     // before divisor. This avoids a momentary overspeed when e.g. switching
46     // to a faster source and increasing divisor to compensate.
47     if (div > clock_hw->div)
48         clock_hw->div = div;
49
50     // If switching a glitchless slice (ref or sys) to an aux source, switch
51     // away from aux *first* to avoid passing glitches when changing aux mux.
52     // Assume (!!!) glitchless source 0 is no faster than the aux source.
53     if (has_glitchless_mux(clock) && src ==
54         CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX) {
55         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_REF_CTRL_SRC_BITS);
```

```
42 } clock_num_t;
```

此外，还有一个结构体用于描述时钟发生器寄存器：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/clocks.h 第 100 - 121 行

```
100 typedef struct {
101     _REG_(CLOCKS_CLK_GPOUT0_CTRL_OFFSET) // CLOCKS_CLK_GPOUT0_CTRL
102     // 时钟控制，除辅助源外可动态更改
103     // 0x00100000 [20]      NUDGE        (0) 此信号的一个边缘使相位发生
104     // 偏移...
105     // 0x00030000 [17:16] PHASE        (0x0) 使能信号延迟最多3个周期
106     // 的...
107     // 0x00001000 [12]      DC50         (0) 启用奇数除数的占空比校正
108     // 0x00000800 [11]      ENABLE        (0) 干净启动和停止时钟生成器
109     // 0x00000400 [10]      KILL         (0) 异步终止时钟生成器
110     // 0x000001e0 [8:5]     AUXSRC       (0x0) 选择辅助时钟源，切换时将出现抖动
111     // when switching
112     io_rw_32 ctrl;
113
114     _REG_(CLOCKS_CLK_GPOUT0_DIV_OFFSET) // CLOCKS_CLK_GPOUT0_DIV
115     // 时钟除数，可动态调整
116     // 0xfffffff00 [31:8]    INT          (0x000001) 除数的整数部分，0 表示
117     // 除以 2^16
118     // 0x000000ff [7:0]     FRAC         (0x00) 除数的小数部分
119     io_rw_32 div;
120
121     _REG_(CLOCKS_CLK_GPOUT0_SELECTED_OFFSET) // CLOCKS_CLK_GPOUT0_SELECTED
122     // 指示当前由无毛刺多路复用器选择的 SRC (单热编码)
123     // 0xffffffff [31:0] CLK_GPOUT0_SELECTED (0x00000001) 此片段不包含无毛刺多路复用器 (仅..
124     . 120 io_ro_32 selected;
125
126 } clock_hw_t;
```

配置时钟时，需了解以下信息：

- 时钟源频率
- 时钟源多路复用器/辅助多路复用器的位置
- 期望输出频率

该 SDK 提供了 `clock_configure` 函数用于配置时钟：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第40至133行

```
40 static void clock_configure_internal(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
41                                     uint32_t actual_freq, uint32_t div) {
42     clock_hw_t *clock_hw = &clocks_hw->clk[clock];
43
44     // 如果分频系数增大，先设置分频系数，再设置时钟源。否则先设置时钟源
45     // 再设置分频系数。该方式避免了例如切换
46     // 到更快时钟源且增加分频系数以补偿时钟源出现的瞬时超频现象。
47     if (div > clock_hw->div)
48         clock_hw->div = div;
49
50     // 若切换无毛刺片段（参考时钟或系统时钟）至辅助时钟源时，需先
51     // 退出辅助源，以避免更改辅助多路复用器时产生毛刺。
52     // 假设 (!!!) 无毛刺源0不比辅助源更快。
53     if (has_glitchless_mux(clock) && src ==
54         CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX) {
55         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_REF_CTRL_SRC_BITS);
```

```

54         while (!(clock_hw->selected & 1u))
55             tight_loop_contents();
56     }
57     // If no glitchless mux, cleanly stop the clock to avoid glitches
58     // propagating when changing aux mux. Note it would be a really bad idea
59     // to do this on one of the glitchless clocks (clk_sys, clk_ref).
60     else {
61         // Disable clock. On clk_ref and clk_sys this does nothing,
62         // all other clocks have the ENABLE bit in the same position.
63         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
64         if (configured_freq[clock] > 0) {
65             // Delay for 3 cycles of the target clock, for ENABLE propagation.
66             // Note XOSC_COUNT is not helpful here because XOSC is not
67             // necessarily running, nor is timer...
68             uint delay_cyc = configured_freq[clk_sys] / configured_freq[clock] + 1;
69             busy_wait_at_least_cycles(delay_cyc * 3);
70         }
71     }
72
73     // Set aux mux first, and then glitchless mux if this clock has one
74     hw_write_masked(&clock_hw->ctrl,
75                     (auxsrc << CLOCKS_CLK_SYS_CTRL_AUXSRC_LSB),
76                     CLOCKS_CLK_SYS_CTRL_AUXSRC_BITS
77     );
78
79     if (has_glitchless_mux(clock)) {
80         hw_write_masked(&clock_hw->ctrl,
81                         src << CLOCKS_CLK_REF_CTRL_SRC_LSB,
82                         CLOCKS_CLK_REF_CTRL_SRC_BITS
83         );
84         while (!(clock_hw->selected & (1u << src)))
85             tight_loop_contents();
86     }
87
88     // Enable clock. On clk_ref and clk_sys this does nothing,
89     // all other clocks have the ENABLE bit in the same position.
90     hw_set_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
91
92     // Now that the source is configured, we can trust that the user-supplied
93     // divisor is a safe value.
94     clock_hw->div = div;
95     configured_freq[clock] = actual_freq;
96 }
97
98 bool clock_configure(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq,
99                      uint32_t freq) {
100    assert(src_freq >= freq);
101
102    if (freq > src_freq)
103        return false;
104
105    uint64_t div64 = (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) / freq;
106    uint32_t div, actual_freq;
107    if (div64 >> 32) {
108        // set div to 0 for maximum clock divider
109        div = 0;
110        actual_freq = src_freq >> (32 - CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
111    } else {
112        div = (uint32_t) div64;
113        // on RP2040 only clock divider of 1, or >= 2 are supported
114        if (div < (2u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB)) {
115            div = (1u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
116        }
117        actual_freq = (uint32_t) (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) /

```

```

54         while (!(clock_hw->selected & 1u))
55             tight_loop_contents();
56     }
57     // 如果没有无毛刺复用器，应干净停止时钟以避免
58     // 更改辅助复用器时出现毛刺传播。请注意，
59     // 在无毛刺时钟（如clk_sys、clk_ref）上执行此操作极其不妥。
60     else {
61         // 禁用时钟。对clk_ref和clk_sys无效，
62         // 其他时钟的ENABLE位均处于同一位置。
63         hw_clear_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
64         if (configured_freq[clock] > 0) {
65             // 延迟目标时钟3个周期，以保证ENABLE信号传播。
66             // 注意XOSC_COUNT在此处无效，因为XOSC未必在运行，定时器亦未必...
67             // 必须运行，定时器也是如此...
68             uint delay_cyc = configured_freq[clk_sys] / configured_freq[clock] + 1;
69             busy_wait_at_least_cycles(delay_cyc * 3);
70         }
71     }
72
73     // 先设置辅助多路复用器，如若时钟含无故障多路复用器，则随后设置该多路复用器
74     hw_write_masked(&clock_hw->ctrl,
75                     (auxsrc << CLOCKS_CLK_SYS_CTRL_AUXSRC_LSB),
76                     CLOCKS_CLK_SYS_CTRL_AUXSRC_BITS
77     );
78
79     if (has_glitchless_mux(clock)) {
80         hw_write_masked(&clock_hw->ctrl,
81                         src << CLOCKS_CLK_REF_CTRL_SRC_LSB,
82                         CLOCKS_CLK_REF_CTRL_SRC_BITS
83         );
84         while (!(clock_hw->selected & (1u << src)))
85             tight_loop_contents();
86     }
87
88     // 启用时钟。对clk_ref和clk_sys无效，
89     // 所有其他时钟的ENABLE位均位于相同位置。
90     hw_set_bits(&clock_hw->ctrl, CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS);
91
92     // 既然源已配置完毕，我们可以信赖用户提供的
93     // 除数为安全值。
94     clock_hw->div = div;
95     configured_freq[clock] = actual_freq;
96 }
97
98 bool clock_configure(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t src_freq,
99                      uint32_t freq) {
100    assert(src_freq >= freq);
101
102    if (freq > src_freq)
103        return false;
104
105    uint64_t div64 = (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) / freq;
106    uint32_t div, actual_freq;
107    if (div64 > 32) {
108        // 将除数设置为0，表示最大时钟分频
109        div = 0;
110        actual_freq = src_freq >> (32 - CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
111    } else {
112        div = (uint32_t) div64;
113        // 仅在RP2040上支持时钟分频为1或不小于2的设置
114        if (div < (2u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB)) {
115            div = (1u << CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
116        }
117        actual_freq = (uint32_t) (((uint64_t) src_freq) << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) /

```

```

        div);
117    }
118
119    clock_configure_internal(clock, src, auxsrc, actual_freq, div);
120    // Store the configured frequency
121    return true;
122 }
123
124 void clock_configure_int_divider(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
125     uint32_t src_freq, uint32_t int_divider) {
126     clock_configure_internal(clock, src, auxsrc, src_freq / int_divider, int_divider <<
127     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
128 }
129
128 void clock_configure_undivided(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t
129     src_freq) {
130     clock_configure_internal(clock, src, auxsrc, src_freq, 1u <<
131     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
132 }

```

It is called in `clocks_init` for each clock. The following example shows the `clk_sys` configuration:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_runtime_init/runtime_init_clocks.c Lines 100 - 104

```

100     // CLK_SYS = PLL_SYS (usually) 125MHz / 1 = 125MHz
101     clock_configure_undivided(clk_sys,
102         CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
103         CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS,
104         SYS_CLK_HZ);

```

Once a clock is configured, `clock_get_hz` can be called to get the output frequency in Hz.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c Lines 137 - 139

```

137 uint32_t clock_get_hz(clock_handle_t clock) {
138     return configured_freq[clock];
139 }

```

WARNING

It is assumed the source frequency the programmer provides is correct. If it is not then the frequency returned by `clock_get_hz` will be inaccurate.

2.15.6.2. Using the frequency counter

To use the frequency counter, the programmer must:

- Set the reference frequency: `clk_ref`
- Set the mux position of the source they want to measure. See [FC0_SRC](#)
- Wait for the `DONE` status bit in [FC0_STATUS](#) to be set
- Read the result

The SDK defines a `frequency_count` function which takes the source as an argument and returns the frequency in kHz:

```

        div);
117    }
118
119    clock_configure_internal(clock, src, auxsrc, actual_freq, div);
120    // 存储已配置的频率
121    return true;
122 }
123
124 void clock_configure_int_divider(clock_handle_t clock, uint32_t src, uint32_t auxsrc,
125     uint32_t src_freq, uint32_t int_divider) {
126     clock_configure_internal(clock, src, auxsrc, src_freq / int_divider, int_divider <<
127     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
128 }
129
128 void clock_configure_undivided(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t
129     src_freq) {
130     clock_configure_internal(clock, src, auxsrc, src_freq, 1u <<
131     CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
132 }

```

该函数在`clocks_init`中针对每个时钟调用。以下示例展示了`clk_sys`的配置：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_runtime_init/runtime_init_clocks.c 第100至104行

```

100     // CLK SYS = PLL SYS (通常) 125MHz / 1 = 125MHz
101     clock_configure_undivided(clk_sys,
102         CLOCKSYS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
103         CLOCKSYS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS,
104         SYS_CLK_HZ);

```

时钟配置完成后，可调用`clock_get_hz`获取输出频率（单位 Hz）。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第137-139行

```

137 uint32_t clock_get_hz(clock_handle_t clock) {
138     return configured_freq[clock];
139 }

```

● 警告

假定程序员提供的源频率是正确的。若不正确，则`clock_get_hz`返回的频率将不准确。

2.15.6.2. 频率计的使用

使用频率计时，程序员必须：

- 设置参考频率：`clk_ref`
- 设置欲测量的信号源的复用器位置。详见`FC0_SRC`
- 等待`FC0_STATUS`中`DONE`状态位被置位
- 读取测量结果

SDK定义了一个`frequency_count`函数，参数为信号源，返回以kHz为单位的频率：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c Lines 147 - 174

```

147 uint32_t frequency_count_khz(uint src) {
148     fc_hw_t *fc = &clocks_hw->fc0;
149
150     // If frequency counter is running need to wait for it. It runs even if the source is NULL
151     while(fc->status & CLOCKS_FC0_STATUS_RUNNING_BITS) {
152         tight_loop_contents();
153     }
154
155     // Set reference freq
156     fc->ref_khz = clock_get_hz(clk_ref) / 1000;
157
158     // FIXME: Don't pick random interval. Use best interval
159     fc->interval = 10;
160
161     // No min or max
162     fc->min_khz = 0;
163     fc->max_khz = 0xffffffff;
164
165     // Set SRC which automatically starts the measurement
166     fc->src = src;
167
168     while(!(fc->status & CLOCKS_FC0_STATUS_DONE_BITS)) {
169         tight_loop_contents();
170     }
171
172     // Return the result
173     return fc->result >> CLOCKS_FC0_RESULT_KHZ_LSB;
174 }
```

There is also a wrapper function to change the unit to MHz:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware/clocks.h Lines 377 - 379

```

377 static inline float frequency_count_mhz(uint src) {
378     return ((float) (frequency_count_khz(src))) / KHZ;
379 }
```

NOTE

The frequency counter can also be used in a test mode. This allows the hardware to check if the frequency is within a minimum frequency and a maximum frequency, set in `FC0_MIN_KHZ` and `FC0_MAX_KHZ`. In this mode, the `PASS` bit in `FC0_STATUS` will be set when `DONE` is set if the frequency is within the specified range. Otherwise, either the `FAST` or `SLOW` bit will be set.

If the programmer attempts to count a stopped clock, or the clock stops running then the `DIED` bit will be set. If any of `DIED`, `FAST`, or `SLOW` are set then `FAIL` will be set.

2.15.6.3. Configuring a GPIO output clock

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c Lines 245 - 276

```

245 void clock_gpio_init_int_frac16(uint gpio, uint src, uint32_t div_int, uint16_t div_frac16)
{
246     // Bit messy but it's as much code to loop through a lookup
247     // table. The sources for each gput generators are the same
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第147-174行

```

147 uint32_t frequency_count_khz(uint src) {
148     fc_hw_t *fc = &clocks_hw->fc0;
149
150 // 如果频率计正在运行，需等待其完成。即使源为NULL，频率计仍然运行 151 while(fc->status & CLOCKS_FC0_STATUS_RUNNING_BITS) {
152     tight_loop_contents();
153 }
154
155 // 设置参考频率
156 fc->ref_khz = clock_get_hz(clk_ref) / 1000;
157
158 // 待修正：不要选择随机间隔。应使用最佳间隔 159 fc->interval = 10;
159
160
161 // 无最小或最大值限制
162 fc->min_khz = 0;
163 fc->max_khz = 0xffffffff;
164
165 // 设置 SRC，自动开始测量
166 fc->src = src;
167
168 while(!(fc->status & CLOCKS_FC0_STATUS_DONE_BITS)) {
169     tight_loop_contents();
170 }
171
172 // 返回结果
173 return fc->result >> CLOCKS_FC0_RESULT_KHZ_LSB;
174 }
```

还提供了一个包装函数，用于将单位转换为MHz：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware_clocks.h 第377至379行

```

377 static inline float frequency_count_mhz(uint src) {
378     return ((float)(frequency_count_khz(src))) / KHZ;
379 }
```

注意

频率计数器亦可在测试模式下使用。该模式允许硬件检测频率是否位于FC0_MIN_KHZ与FC0_MAX_KHZ设定的最小与最大频率之间。当频率在指定范围内且DONE位被置位时，FC0_STATUS的PASS位将被设置。否则，将设置FAST位或SLOW位。

如果程序员尝试读取已停止的时钟，或时钟停止运行，则会设置 DIED位。若DIED、FAST 或SLOW中任一标志被置位，则FAIL标志也将被设置。

2.15.6.3. 配置GPIO输出时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第245至276行

```

245 void clock_gpio_init_int_frac16(uint gpio, uint src, uint32_t div_int, uint16_t div_frac16)
{
246     // 代码较为杂乱，但遍历查找表的代码量相当
247     // 每个gout发生器的时钟源均相同
```

```

248     // so just call with the sources from GP0
249     uint gpclk = 0;
250     if      (gpio == 21) gpclk = clk_gpout0;
251     else if (gpio == 23) gpclk = clk_gpout1;
252     else if (gpio == 24) gpclk = clk_gpout2;
253     else if (gpio == 25) gpclk = clk_gpout3;
254     else {
255         invalid_params_if(HARDWARE_CLOCKS, true);
256     }
257
258     invalid_params_if(HARDWARE_CLOCKS, div_int >> REG_FIELD_WIDTH(
259         CLOCKS_CLK_GPOUT0_DIV_INT));
260     // Set up the gpclk generator
261     clocks_hw->clk[gpclk].ctrl = (src << CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_LSB) |
262                                     CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS;
263 #ifdef REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 16
264     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | (div_frac16 <<
265                                     CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
266 #elif REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 8
267     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | ((div_frac16
268         >>8u) << CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
269 #else
270     //error unsupported number of fractional bits
271 #endif
272     // Set gpio pin to gpclock function
273     gpio_set_function(gpio, GPIO_FUNC_GPCK);
274 }
```

2.15.6.4. Configuring a GPIO input clock

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c Lines 313 - 343

```

313 bool clock_configure_gpin(clock_handle_t clock, uint gpio, uint32_t src_freq, uint32_t freq)
{
314     // Configure a clock to run from a GPIO input
315     uint gpin = 0;
316     if      (gpio == 20) gpin = 0;
317     else if (gpio == 22) gpin = 1;
318     else {
319         invalid_params_if(HARDWARE_CLOCKS, true);
320     }
321
322     // Work out sources. GPIN is always an auxsrc
323     uint src = 0;
324
325     // GPIN1 == GPIN0 + 1
326     uint auxsrc = gpin0_src[clock] + gpin;
327
328     if (has_glitchless_mux(clock)) {
329         // AUX src is always 1
330         src = 1;
331     }
332
333     // Set the GPIO function
334     gpio_set_function(gpio, GPIO_FUNC_GPCK);
335
336     // Now we have the src, auxsrc, and configured the gpio input
337     // call clock configure to run the clock from a gpio
338     return clock_configure(clock, src, auxsrc, src_freq, freq);
```

```

248 // 因此仅需使用GP0的时钟源调用
249     uint gpclk = 0;
250     if (gpio == 21) gpclk = clk_gpout0;
251     else if (gpio == 23) gpclk = clk_gpout1;
252     else if (gpio == 24) gpclk = clk_gpout2;
253     else if (gpio == 25) gpclk = clk_gpout3;
254     else {
255         invalid_params_if(HARDWARE_CLOCKS, true);
256     }
257
258     invalid_params_if(HARDWARE_CLOCKS, div_int >> REG_FIELD_WIDTH(
259         CLOCKS_CLK_GPOUT0_DIV_INT));
260     // 设置 gpclk 生成器
261     clocks_hw->clk[gpclk].ctrl = (src << CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_LSB) |
262                                     CLOCKS_CLK_GPOUT0_CTRL_ENABLE_BITS;
263 #ifdef REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 16
264     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | (div_frac16 <<
265         CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
266 #elif REG_FIELD_WIDTH(CLOCKS_CLK_GPOUT0_DIV_FRAC) == 8
267     clocks_hw->clk[gpclk].div = (div_int << CLOCKS_CLK_GPOUT0_DIV_INT_LSB) | ((div_frac16
268         >>8u) << CLOCKS_CLK_GPOUT0_DIV_FRAC_LSB);
269 #else
270     #error 不支持的分位数
271 #endif
272     // 设置 GPIO 引脚为 gpclock 功能
273     gpio_set_function(gpio, GPIO_FUNC_GPCK);
274 }
```

2.15.6.4. 配置 GPIO 输入时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第 313-343 行

```

313 bool clock_configure_gpin(clock_handle_t clock, uint gpio, uint32_t src_freq, uint32_t freq)
{
314     // 配置时钟以从 GPIO 输入运行
315     uint gpin = 0;
316     if (gpio == 20) gpin = 0;
317     else if (gpio == 22) gpin = 1;
318     else {
319         invalid_params_if(HARDWARE_CLOCKS, true);
320     }
321
322 // 计算信号源GPIN 始终为 auxsrc 323 uint src = 0;

324
325 // GPIN1 == GPIN0 + 1
326     uint auxsrc = gpin0_src[clock] + gpin;
327
328     if (has_glitchless_mux(clock)) {
329         // AUX 源始终为 1
330         src = 1;
331     }
332
333 // 设置 GPIO 功能
334     gpio_set_function(gpio, GPIO_FUNC_GPCK);
335
336 // 目前已有 src、auxsrc，并完成 gpio 输入的配置
337 // 调用 clock_configure 以从 gpio 驱动时钟
338     return clock_configure(clock, src, auxsrc, src_freq, freq);
```

339 }

2.15.6.5. Enabling resus

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c Lines 221 - 243

```

221 void clocks_enable_resus(resus_callback_t resus_callback) {
222     // Restart clk_sys if it is stopped by forcing it
223     // to the default source of clk_ref. If clk_ref stops running this will
224     // not work.
225
226     // Store user's resus callback
227     _resus_callback = resus_callback;
228
229     irq_set_exclusive_handler(CLOCKS_IRQ, clocks_irq_handler);
230
231     // Enable the resus interrupt in clocks
232     clocks_hw->inte = CLOCKS_INTE_CLK_SYS_RESUS_BITS;
233
234     // Enable the clocks irq
235     irq_set_enabled(CLOCKS_IRQ, true);
236
237     // 2 * clk_ref freq / clk_sys_min_freq;
238     // assume clk_ref is 3MHz and we want clk_sys to be no lower than 1MHz
239     uint timeout = 2 * 3 * 1;
240
241     // Enable resus with the maximum timeout
242     clocks_hw->resus.ctrl = CLOCKS_CLK_SYS_RESUS_CTRL_ENABLE_BITS | timeout;
243 }
```

2.15.6.6. Configuring sleep mode

Sleep mode is active when neither processor core or the DMA are requesting clocks. For example, the DMA is not active and both core0 and core1 are waiting for an interrupt. The `SLEEP_EN` registers set what clocks are running in sleep mode. The `hello_sleep` example (https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c) illustrates how to put the chip to sleep until the RTC fires. In this case, only the RTC clock is enabled in the `SLEEP_EN0` register.

NOTE

`clk_sys` is always sent to `proc0` and `proc1` during sleep mode as some logic needs to be clocked for the processor to wake up again.

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c Lines 159 - 183

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // We should have already called the sleep_run_from_dormant_source function
163     // This is only needed for dormancy although it saves power running from xosc while
164     // sleeping
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCKS_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;
```

339 }

2.15.6.5. 启用 resus

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/clocks.c 第 221 - 243 行

```

221 void clocks_enable_resus(resus_callback_t resus_callback) {
222     // 如果 clk_sys 被停止，则通过强制其重启
223 // 到 clk_ref 的默认时钟源。如果 clk_ref 停止运行，本方法224 // 将不起作用。
224
225     // 保存用户的 resus 回调函数
226     _resus_callback = resus_callback;
227
228     irq_set_exclusive_handler(CLOCKS_IRQ, clocks_irq_handler);
229
230     // 启用 clocks 中的 resus 中断
231     clocks_hw->inte = CLOCKS_INTE_CLK_SYS_RESUS_BITS;
232
233     // 启用 clocks 中断
234     irq_set_enabled(CLOCKS_IRQ, true);
235
236     // 2 * clk_ref 频率 / clk_sys_min_freq;
237     // 假设 clk_ref 为 3MHz，且期望 clk_sys 不低于 1MHz
238     uint timeout = 2 * 3 * 1;
239
240     // 使用最大超时时间启用复苏功能
241     clocks_hw->resus.ctrl = CLOCKS_CLK_SYS_RESUS_CTRL_ENABLE_BITS | timeout;
242
243 }

```

2.15.6.6 配置睡眠模式

当处理器核心和 DMA 均未请求时钟时，睡眠模式即处于激活状态。例如，当 DMA 不活动且 core0与 core1均在等待中断时。寄存器 SLEEP_EN用于设置睡眠模式下运行的时钟。

[示例程序hello_sleep](#) (https://github.com/raspberrypi/pico-playground/blob/master/sleep/hello_sleep/hello_sleep_aon.c) 展示了如何使芯片进入睡眠状态，直至 RTC 触发。在此情况下，仅在 SLEEP_EN0 寄存器中启用 RTC 时钟。

① 注意

clk_sys 在睡眠模式期间始终发送至 proc0 和 proc1，因某些逻辑需时钟驱动以使处理器能够重新唤醒。

Pico 附加功能: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/pico_sleep/sleep.c 第159至183行

```

159 void sleep_goto_sleep_until(struct timespec *ts, aon_timer_alarm_handler_t callback)
160 {
161
162     // 我们应已调用过 sleep_run_from_dormant_source 函数
163 // 该功能仅在休眠状态下需要，尽管它在睡眠时通过 xosc 节约了功耗
164
165     //assert(dormant_source_valid(_dormant_source));
166
167     clocks_hw->sleep_en0 = CLOCKS_SLEEP_EN0_CLK_RTC_RTC_BITS;
168     clocks_hw->sleep_en1 = 0x0;

```

```

168
169     aon_timer_enable_alarm(ts, callback, false);
170
171     stdio_flush();
172
173     // Enable deep sleep at the proc
174     processor_deep_sleep();
175
176     // Go to sleep
177     __wfi();
178 }
```

2.15.7. List of Registers

The Clocks registers start at a base address of **0x40008000** (defined as **CLOCKS_BASE** in SDK).

Table 207. List of CLOCKS registers

Offset	Name	Info
0x00	CLK_GPOUT0_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x04	CLK_GPOUT0_DIV	Clock divisor, can be changed on-the-fly
0x08	CLK_GPOUT0_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x0c	CLK_GPOUT1_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x10	CLK_GPOUT1_DIV	Clock divisor, can be changed on-the-fly
0x14	CLK_GPOUT1_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x18	CLK_GPOUT2_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x1c	CLK_GPOUT2_DIV	Clock divisor, can be changed on-the-fly
0x20	CLK_GPOUT2_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x24	CLK_GPOUT3_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x28	CLK_GPOUT3_DIV	Clock divisor, can be changed on-the-fly
0x2c	CLK_GPOUT3_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x30	CLK_REF_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x34	CLK_REF_DIV	Clock divisor, can be changed on-the-fly
0x38	CLK_REF_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x3c	CLK_SYS_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x40	CLK_SYS_DIV	Clock divisor, can be changed on-the-fly
0x44	CLK_SYS_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x48	CLK_PERI_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x50	CLK_PERI_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).

```

168
169     aon_timer_enable_alarm(ts, callback, false);
170
171     stdio_flush();
172
173     // 在处理器上启用深度睡眠
174     processor_deep_sleep();
175
176     // 进入睡眠状态
177     __wfi();
178 }

```

2.15.7. 寄存器列表

时钟寄存器地址为 `0x40008000`（在 SDK 中定义为 CLOCKS_BASE）。

表207. CLOCKS
寄存器列表

偏移量	名称	说明
0x00	<code>CLK_GPOUT0_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x04	<code>CLK_GPOUT0_DIV</code>	时钟分频器，支持动态更改
0x08	<code>CLK_GPOUT0_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x0c	<code>CLK_GPOUT1_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x10	<code>CLK_GPOUT1_DIV</code>	时钟分频器，支持动态更改
0x14	<code>CLK_GPOUT1_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x18	<code>CLK_GPOUT2_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x1c	<code>CLK_GPOUT2_DIV</code>	时钟分频器，支持动态更改
0x20	<code>CLK_GPOUT2_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x24	<code>CLK_GPOUT3_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x28	<code>CLK_GPOUT3_DIV</code>	时钟分频器，支持动态更改
0x2c	<code>CLK_GPOUT3_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x30	<code>CLK_REF_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x34	<code>CLK_REF_DIV</code>	时钟分频器，支持动态更改
0x38	<code>CLK_REF_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x3c	<code>CLK_SYS_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x40	<code>CLK_SYS_DIV</code>	时钟分频器，支持动态更改
0x44	<code>CLK_SYS_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x48	<code>CLK_PERI_CTRL</code>	时钟控制，支持动态更改（auxsrc 除外）
0x50	<code>CLK_PERI_SELECTED</code>	指示防毛刺多路复用器（单热编码）当前选择的 SRC。

Offset	Name	Info
0x54	CLK_USB_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x58	CLK_USB_DIV	Clock divisor, can be changed on-the-fly
0x5c	CLK_USB_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x60	CLK_ADC_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x64	CLK_ADC_DIV	Clock divisor, can be changed on-the-fly
0x68	CLK_ADC_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x6c	CLK_RTC_CTRL	Clock control, can be changed on-the-fly (except for auxsrc)
0x70	CLK_RTC_DIV	Clock divisor, can be changed on-the-fly
0x74	CLK_RTC_SELECTED	Indicates which SRC is currently selected by the glitchless mux (one-hot).
0x78	CLK_SYS_RESUS_CTRL	
0x7c	CLK_SYS_RESUS_STATUS	
0x80	FC0_REF_KHZ	Reference clock frequency in kHz
0x84	FC0_MIN_KHZ	Minimum pass frequency in kHz. This is optional. Set to 0 if you are not using the pass/fail flags
0x88	FC0_MAX_KHZ	Maximum pass frequency in kHz. This is optional. Set to 0xffffffff if you are not using the pass/fail flags
0x8c	FC0_DELAY	Delays the start of frequency counting to allow the mux to settle Delay is measured in multiples of the reference clock period
0x90	FC0_INTERVAL	The test interval is 0.98us * 2**interval, but let's call it 1us * 2**interval The default gives a test interval of 250us
0x94	FC0_SRC	Clock sent to frequency counter, set to 0 when not required Writing to this register initiates the frequency count
0x98	FC0_STATUS	Frequency counter status
0x9c	FC0_RESULT	Result of frequency measurement, only valid when status_done=1
0xa0	WAKE_EN0	enable clock in wake mode
0xa4	WAKE_EN1	enable clock in wake mode
0xa8	SLEEP_EN0	enable clock in sleep mode
0xac	SLEEP_EN1	enable clock in sleep mode
0xb0	ENABLED0	indicates the state of the clock enable
0xb4	ENABLED1	indicates the state of the clock enable
0xb8	INTR	Raw Interrupts
0xbc	INTE	Interrupt Enable
0xc0	INTF	Interrupt Force
0xc4	INTS	Interrupt status after masking & forcing

偏移量	名称	说明
0x54	CLK_USB_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x58	CLK_USB_DIV	时钟分频器，支持动态更改
0x5c	CLK_USB_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x60	CLK_ADC_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x64	CLK_ADC_DIV	时钟分频器，支持动态更改
0x68	CLK_ADC_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x6c	CLK_RTC_CTRL	时钟控制，支持动态更改（auxsrc 除外）
0x70	CLK_RTC_DIV	时钟分频器，支持动态更改
0x74	CLK_RTC_SELECTED	指示防毛刺多路复用器（单热编码）当前选择的 SRC。
0x78	CLK_SYS_RESUS_CTRL	
0x7c	CLK_SYS_RESUS_STATUS	
0x80	FC0_REF_KHZ	参考时钟频率，单位 kHz
0x84	FC0_MIN_KHZ	最小通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0。
0x88	FC0_MAX_KHZ	最大通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0xffffffff。
0x8c	FC0_DELAY	延迟频率计数启动，以使多路复用器稳定 延迟以参考时钟周期的整数倍计量
0x90	FC0_INTERVAL	测试间隔为0.98微秒 * 2**interval，但我们称之为1微秒 * 2* *interval 默认测试间隔为250微秒
0x94	FC0_SRC	传送至频率计数器的时钟，非必要时设置为0 写入此寄存器即可启动频率计数
0x98	FC0_STATUS	频率计数器状态
0x9c	FC0_RESULT	频率测量结果，仅在status_done=1时有效
0xa0	WAKE_EN0	启用唤醒模式下的时钟
0xa4	WAKE_EN1	启用唤醒模式下的时钟
0xa8	SLEEP_EN0	启用睡眠模式下的时钟
0xac	SLEEP_EN1	启用睡眠模式下的时钟
0xb0	ENABLED0	指示时钟使能状态
0xb4	ENABLED1	指示时钟使能状态
0xb8	INTR	原始中断
0xbc	INTE	中断使能
0xc0	INTF	中断强制
0xc4	INTS	掩码及强制后的中断状态

CLOCKS: CLK_GPOUT0_CTRL Register

Offset: 0x00

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 208.
CLK_GPOUT0_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUDGE : An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE : This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:13	Reserved.	-	-
12	DC50 : Enables duty cycle correction for odd divisors	RW	0x0
11	ENABLE : Starts and stops the clock generator cleanly	RW	0x0
10	KILL : Asynchronously kills the clock generator	RW	0x0
9	Reserved.	-	-
8:5	AUXSRC : Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIO1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	Reserved.	-	-

CLOCKS: CLK_GPOUT0_DIV Register

Offset: 0x04

Description

Clock divisor, can be changed on-the-fly

CLOCKS: CLK_GPOUT0_CTRL 寄存器

偏移: 0x00

描述

时钟控制，支持动态更改（auxsrc 除外）

表208
CLK_GPOUT0_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_SYS 0x1 → CLKSRC_GPIO 0x2 → CLKSRC_GPIO1 0x3 → CLKSRC_PLL_USB 0x4 → ROSC_CLKSRC 0x5 → XOSC_CLKSRC 0x6 → CLK_SYS 0x7 → CLK_USB 0x8 → CLK_ADC 0x9 → CLK_RTC 0xa → CLK_REF	读写	0x0
4:0	保留。	-	-

CLOCKS: CLK_GPOUT0_DIV 寄存器

偏移量: 0x04

说明

时钟分频器，支持动态更改

Table 209.
CLK_GPOUT0_DIV
Register

Bits	Description	Type	Reset
31:8	INT : Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	FRAC : Fractional component of the divisor	RW	0x00

CLOCKS: CLK_GPOUT0_SELECTED Register

Offset: 0x08

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 210.
CLK_GPOUT0_SELECT
ED Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_GPOUT1_CTRL Register

Offset: 0x0c

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 211.
CLK_GPOUT1_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUDGE : An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE : This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:13	Reserved.	-	-
12	DC50 : Enables duty cycle correction for odd divisors	RW	0x0
11	ENABLE : Starts and stops the clock generator cleanly	RW	0x0
10	KILL : Asynchronously kills the clock generator	RW	0x0
9	Reserved.	-	-
8:5	AUXSRC : Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIO1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		

表 209。
CLK_GPOUT0_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分, 0 —除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT0_SELECTED 寄存器

偏移量: 0x08

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 210。
CLK_GPOUT0_SELEC
T_ED 寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT1_CTRL 寄存器

偏移: 0x0c

说明

时钟控制，支持动态更改（auxsrc 除外）

表 211。
CLK_GPOUT1_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_SYS 0x1 → CLKSRC_GPIO 0x2 → CLKSRC_GPIO1 0x3 → CLKSRC_PLL_USB 0x4 → ROSC_CLKSRC 0x5 → XOSC_CLKSRC 0x6 → CLK_SYS 0x7 → CLK_USB	读写	0x0

Bits	Description	Type	Reset
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	Reserved.	-	-

CLOCKS: CLK_GPOUT1_DIV Register

Offset: 0x10

Description

Clock divisor, can be changed on-the-fly

Table 212.
CLK_GPOUT1_DIV
Register

Bits	Description	Type	Reset
31:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	FRAC: Fractional component of the divisor	RW	0x00

CLOCKS: CLK_GPOUT1_SELECTED Register

Offset: 0x14

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 213.
CLK_GPOUT1_SELECT
ED Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_GPOUT2_CTRL Register

Offset: 0x18

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 214.
CLK_GPOUT2_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUDGE: An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE: This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:13	Reserved.	-	-
12	DC50: Enables duty cycle correction for odd divisors	RW	0x0
11	ENABLE: Starts and stops the clock generator cleanly	RW	0x0
10	KILL: Asynchronously kills the clock generator	RW	0x0
9	Reserved.	-	-

位	描述	类型	复位值
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT1_DIV 寄存器

偏移: 0x10

说明

时钟分频器，支持动态更改

表 212
CLK_GPOUT1_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT1_SELECTED 寄存器

偏移: 0x14

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 213
CLK_GPOUT1_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT2_CTRL 寄存器

偏移: 0x18

说明

时钟控制，支持动态更改（auxsrc 除外）

表 214
CLK_GPOUT2_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-

Bits	Description	Type	Reset
8:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIN0		
	0x2 → CLKSRC_GPIN1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	Reserved.	-	-

CLOCKS: CLK_GPOUT2_DIV Register

Offset: 0x1c

Description

Clock divisor, can be changed on-the-fly

Table 215.
CLK_GPOUT2_DIV
Register

Bits	Description	Type	Reset
31:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	FRAC: Fractional component of the divisor	RW	0x00

CLOCKS: CLK_GPOUT2_SELECTED Register

Offset: 0x20

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 216.
CLK_GPOUT2_SELECT
ED Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_GPOUT3_CTRL Register

Offset: 0x24

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 217.
CLK_GPOUT3_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-

位	描述	类型	复位值
8:5	AUXSRC: 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIN0		
	0x2 → CLKSRC_GPIN1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT2_DIV 寄存器

偏移：0x1c

说明

时钟分频器，支持动态更改

表 215
CLK_GPOUT2_DIV
寄存器

位	描述	类型	复位值
31:8	INT: 除数的整数部分，0 → 除以 2^{16}	读写	0x000001
7:0	FRAC: 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT2_SELECTED 寄存器

偏移：0x20

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 216
CLK_GPOUT2_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_GPOUT3_CTRL 寄存器

偏移：0x24

说明

时钟控制，支持动态更改（auxsrc 除外）

表 217。
CLK_GPOUT3_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-

Bits	Description	Type	Reset
20	NUDGE: An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE: This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:13	Reserved.	-	-
12	DC50: Enables duty cycle correction for odd divisors	RW	0x0
11	ENABLE: Starts and stops the clock generator cleanly	RW	0x0
10	KILL: Asynchronously kills the clock generator	RW	0x0
9	Reserved.	-	-
8:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIN1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	Reserved.	-	-

CLOCKS: CLK_GPOUT3_DIV Register

Offset: 0x28

Description

Clock divisor, can be changed on-the-fly

Table 218.
CLK_GPOUT3_DIV
Register

Bits	Description	Type	Reset
31:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	FRAC: Fractional component of the divisor	RW	0x00

CLOCKS: CLK_GPOUT3_SELECTED Register

Offset: 0x2c

位	描述	类型	复位值
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:13	保留。	-	-
12	DC50 : 启用奇数除数的占空比校正	读写	0x0
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9	保留。	-	-
8:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_GPIN0		
	0x2 → CLKSRC_GPIN1		
	0x3 → CLKSRC_PLL_USB		
	0x4 → ROSC_CLKSRC_PH		
	0x5 → XOSC_CLKSRC		
	0x6 → CLK_SYS		
	0x7 → CLK_USB		
	0x8 → CLK_ADC		
	0x9 → CLK_RTC		
	0xa → CLK_REF		
4:0	保留。	-	-

CLOCKS: CLK_GPOUT3_DIV寄存器

偏移: 0x28

描述

时钟分频器，支持动态更改

表218。
CLK_GPOUT3_DIV
寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 – 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_GPOUT3_SELECTED寄存器

偏移: 0x2c

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 219.
CLK_GPOUT3_SELECT
ED Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_REF_CTRL Register

Offset: 0x30

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 220.
CLK_REF_CTRL
Register

Bits	Description	Type	Reset
31:7	Reserved.	-	-
6:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIO1		
4:2	Reserved.	-	-
1:0	SRC: Selects the clock source glitchlessly, can be changed on-the-fly	RW	-
	Enumerated values:		
	0x0 → ROSC_CLKSRC_PH		
	0x1 → CLKSRC_CLK_REF_AUX		
	0x2 → XOSC_CLKSRC		

CLOCKS: CLK_REF_DIV Register

Offset: 0x34

Description

Clock divisor, can be changed on-the-fly

Table 221.
CLK_REF_DIV Register

Bits	Description	Type	Reset
31:10	Reserved.	-	-
9:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x1
7:0	Reserved.	-	-

CLOCKS: CLK_REF_SELECTED Register

Offset: 0x38

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表219。
CLK_GPOUT3_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_REF_CTRL 寄存器

偏移：0x30

描述

时钟控制，支持动态更改（auxsrc 除外）

表220。
CLK_REF_CTRL
寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6:5	AUXSRC ：选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_GPIO		
	0x2 → CLKSRC_GPIO1		
4:2	保留。	-	-
1:0	SRC ：无毛刺方式选择时钟源，支持动态切换	读写	-
	枚举值：		
	0x0 → ROOSC_CLKSRC_PH		
	0x1 → CLKSRC_CLK_REF_AUX		
	0x2 → XOSC_CLKSRC		

CLOCKS: CLK_REF_DIV 寄存器

偏移：0x34

描述

时钟分频器，支持动态更改

表 221。
CLK_REF_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT ：除数的整数部分，0 – 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_REF_SELECTED 寄存器

偏移：0x38

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

Table 222.
CLK_REF_SELECTED
Register

Bits	Description	Type	Reset
31:0	The glitchless multiplexer does not switch instantaneously (to avoid glitches), so software should poll this register to wait for the switch to complete. This register contains one decoded bit for each of the clock sources enumerated in the CTRL_SRC field. At most one of these bits will be set at any time, indicating that clock is currently present at the output of the glitchless mux. Whilst switching is in progress, this register may briefly show all-0s.	RO	0x00000001

CLOCKS: CLK_SYS_CTRL Register

Offset: 0x3c

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 223.
CLK_SYS_CTRL
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_PLL_USB		
	0x2 → ROSC_CLKSRC		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:1	Reserved.	-	-
0	SRC: Selects the clock source glitchlessly, can be changed on-the-fly	RW	0x0
	Enumerated values:		
	0x0 → CLK_REF		
	0x1 → CLKSRC_CLK_SYS_AUX		

CLOCKS: CLK_SYS_DIV Register

Offset: 0x40

Description

Clock divisor, can be changed on-the-fly

Table 224.
CLK_SYS_DIV Register

Bits	Description	Type	Reset
31:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	FRAC: Fractional component of the divisor	RW	0x00

CLOCKS: CLK_SYS_SELECTED Register

Offset: 0x44

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

表 222。
CLK_REF_SELECTED
寄存器

位	描述	类型	复位值
31:0	无毛刺多路复用器的切换不是瞬时完成的（以避免毛刺），因此软件应轮询该寄存器以等待切换完成。该寄存器包含 CTRL_SRC 字段中枚举的每个时钟源的一个解码位。任一时刻最多只有一个位被置位，表示该时钟当前在无毛刺多路复用器的输出端存在。在切换过程中，该寄存器可能短暂显示全零。	只读	0x00000001

CLOCKS: CLK_SYS_CTRL 寄存器

偏移: 0x3c

描述

时钟控制，支持动态更改（auxsrc 除外）

表 223。
CLK_SYS_CTRL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_SYS		
	0x1 → CLKSRC_PLL_USB		
	0x2 → ROSC_CLKSRC		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:1	保留。	-	-
0	SRC : 无毛刺方式选择时钟源，支持动态切换	读写	0x0
	枚举值：		
	0x0 → CLK_REF		
	0x1 → CLKSRC_CLK_SYS_AUX		

CLOCKS: CLK_SYS_DIV 寄存器

偏移: 0x40

描述

时钟分频器，支持动态更改

表 224
CLK_SYS_DIV 寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分，0 – 除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_SYS_SELECTED 寄存器

偏移: 0x44

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

Table 225.
CLK_SYS_SELECTED
Register

Bits	Description	Type	Reset
31:0	The glitchless multiplexer does not switch instantaneously (to avoid glitches), so software should poll this register to wait for the switch to complete. This register contains one decoded bit for each of the clock sources enumerated in the CTRL_SRC field. At most one of these bits will be set at any time, indicating that clock is currently present at the output of the glitchless mux. Whilst switching is in progress, this register may briefly show all-0s.	RO	0x00000001

CLOCKS: CLK_PERI_CTRL Register

Offset: 0x48

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 226.
CLK_PERI_CTRL
Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	ENABLE : Starts and stops the clock generator cleanly	RW	0x0
10	KILL : Asynchronously kills the clock generator	RW	0x0
9:8	Reserved.	-	-
7:5	AUXSRC : Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLK_SYS		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → CLKSRC_PLL_USB		
	0x3 → ROSC_CLKSRC_PH		
	0x4 → XOSC_CLKSRC		
	0x5 → CLKSRC_GPI0		
	0x6 → CLKSRC_GPIN1		
4:0	Reserved.	-	-

CLOCKS: CLK_PERI_SELECTED Register

Offset: 0x50

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 227.
CLK_PERI_SELECTED
Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_USB_CTRL Register

Offset: 0x54

Description

Clock control, can be changed on-the-fly (except for auxsrc)

表 225
CLK_SYS_SELECTED
寄存器

位	描述	类型	复位值
31:0	无毛刺多路复用器的切换不是瞬时完成的（以避免毛刺），因此软件应轮询该寄存器以等待切换完成。该寄存器包含 CTRL_SRC 字段中枚举的每个时钟源的一个解码位。任一时刻最多只有一个位被置位，表示该时钟当前在无毛刺多路复用器的输出端存在。在切换过程中，该寄存器可能短暂显示全零。	只读	0x00000001

CLOCKS: CLK_PERI_CTRL 寄存器

偏移: 0x48

描述

时钟控制，支持动态更改（auxsrc 除外）

表 226
CLK_PERI_CTRL
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLK_SYS		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → CLKSRC_PLL_USB		
	0x3 → ROSC_CLKSRC_PH		
	0x4 → XOSC_CLKSRC		
	0x5 → CLKSRC_GPIN0		
	0x6 → CLKSRC_GPIN1		
4:0	保留。	-	-

时钟：CLK_PERI_SELECTED 寄存器

偏移：0x50

说明

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 227.
CLK_PERI_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

时钟：CLK_USB_CTRL 寄存器

偏移：0x54

说明

时钟控制，支持动态更改（auxsrc 除外）

Table 228.
CLK_USB_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUDGE : An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE : This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:12	Reserved.	-	-
11	ENABLE : Starts and stops the clock generator cleanly	RW	0x0
10	KILL : Asynchronously kills the clock generator	RW	0x0
9:8	Reserved.	-	-
7:5	AUXSRC : Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	Reserved.	-	-

CLOCKS: CLK_USB_DIV Register

Offset: 0x58

Description

Clock divisor, can be changed on-the-fly

Table 229.
CLK_USB_DIV Register

Bits	Description	Type	Reset
31:10	Reserved.	-	-
9:8	INT : Integer component of the divisor, 0 → divide by 2^{16}	RW	0x1
7:0	Reserved.	-	-

CLOCKS: CLK_USB_SELECTED Register

Offset: 0x5c

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

表 228.
CLK_USB_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	保留。	-	-

时钟：CLK_USB_DIV 寄存器

偏移：0x58

说明

时钟分频器，支持动态更改

表 229.
CLK_USB_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_USB_SELECTED 寄存器

偏移：0x5c

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

Table 230.
CLK_USB_SELECTED
Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_ADC_CTRL Register

Offset: 0x60

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 231.
CLK_ADC_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUUDGE: An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE: This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:12	Reserved.	-	-
11	ENABLE: Starts and stops the clock generator cleanly	RW	0x0
10	KILL: Asynchronously kills the clock generator	RW	0x0
9:8	Reserved.	-	-
7:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	Reserved.	-	-

CLOCKS: CLK_ADC_DIV Register

Offset: 0x64

Description

Clock divisor, can be changed on-the-fly

Table 232.
CLK_ADC_DIV Register

Bits	Description	Type	Reset
31:10	Reserved.	-	-
9:8	INT: Integer component of the divisor, 0 → divide by 2^{16}	RW	0x1
7:0	Reserved.	-	-

表 230
CLK_USB_SELECTED 寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_ADC_CTRL 寄存器

偏移: 0x60

描述

时钟控制，支持动态更改（auxsrc 除外）

表 231
CLK_ADC_CTRL 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺	读写	0x0
	枚举值：		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	保留。	-	-

CLOCKS: CLK_ADC_DIV 寄存器

偏移: 0x64

描述

时钟分频器，支持动态更改

表 232
CLK_ADC_DIV 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:8	INT : 除数的整数部分，0 → 除以 2^{16}	读写	0x1
7:0	保留。	-	-

CLOCKS: CLK_ADC_SELECTED Register

Offset: 0x68

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 233.
CLK_ADC_SELECTED
Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_RTC_CTRL Register

Offset: 0x6c

Description

Clock control, can be changed on-the-fly (except for auxsrc)

Table 234.
CLK_RTC_CTRL
Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	NUDGE: An edge on this signal shifts the phase of the output by 1 cycle of the input clock This can be done at any time	RW	0x0
19:18	Reserved.	-	-
17:16	PHASE: This delays the enable signal by up to 3 cycles of the input clock This must be set before the clock is enabled to have any effect	RW	0x0
15:12	Reserved.	-	-
11	ENABLE: Starts and stops the clock generator cleanly	RW	0x0
10	KILL: Asynchronously kills the clock generator	RW	0x0
9:8	Reserved.	-	-
7:5	AUXSRC: Selects the auxiliary clock source, will glitch when switching	RW	0x0
	Enumerated values:		
	0x0 → CLKSRC_PLL_USB		
	0x1 → CLKSRC_PLL_SYS		
	0x2 → ROSC_CLKSRC_PH		
	0x3 → XOSC_CLKSRC		
	0x4 → CLKSRC_GPIN0		
	0x5 → CLKSRC_GPIN1		
4:0	Reserved.	-	-

CLOCKS: CLK_RTC_DIV Register

Offset: 0x70

Description

Clock divisor, can be changed on-the-fly

CLOCKS: CLK_ADC_SELECTED 寄存器

偏移: 0x68

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 233.
CLK_ADC_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_RTC_CTRL 寄存器

偏移: 0x6c

描述

时钟控制，支持动态更改（auxsrc 除外）

表 234.
CLK_RTC_CTRL
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	NUDGE : 该信号的一个边沿将输出相位移动输入时钟的1个周期 此操作可在任何时间进行	读写	0x0
19:18	保留。	-	-
17:16	PHASE : 将使能信号延迟最多3个输入时钟周期 此设置必须在时钟使能前完成，方能生效	读写	0x0
15:12	保留。	-	-
11	ENABLE : 干净地启动和停止时钟发生器	读写	0x0
10	KILL : 异步关闭时钟发生器	读写	0x0
9:8	保留。	-	-
7:5	AUXSRC : 选择辅助时钟源，切换时会产生毛刺 枚举值： 0x0 → CLKSRC_PLL_USB 0x1 → CLKSRC_PLL_SYS 0x2 → ROSC_CLKSRC_PH 0x3 → XOSC_CLKSRC 0x4 → CLKSRC_GPIN0 0x5 → CLKSRC_GPIN1	读写	0x0
4:0	保留。	-	-

CLOCKS: CLK_RTC_DIV 寄存器

偏移: 0x70

描述

时钟分频器，支持动态更改

Table 235.
CLK_RTC_DIV Register

Bits	Description	Type	Reset
31:8	INT : Integer component of the divisor, 0 → divide by 2^{16}	RW	0x000001
7:0	Frac : Fractional component of the divisor	RW	0x00

CLOCKS: CLK_RTC_SELECTED Register

Offset: 0x74

Description

Indicates which SRC is currently selected by the glitchless mux (one-hot).

Table 236.
CLK_RTC_SELECTED
Register

Bits	Description	Type	Reset
31:0	This slice does not have a glitchless mux (only the AUX_SRC field is present, not SRC) so this register is hardwired to 0x1.	RO	0x00000001

CLOCKS: CLK_SYS_RESUS_CTRL Register

Offset: 0x78

Table 237.
CLK_SYS_RESUS_CTRL
Register

Bits	Description	Type	Reset
31:17	Reserved.	-	-
16	CLEAR : For clearing the resus after the fault that triggered it has been corrected	RW	0x0
15:13	Reserved.	-	-
12	FRCE : Force a resus, for test purposes only	RW	0x0
11:9	Reserved.	-	-
8	ENABLE : Enable resus	RW	0x0
7:0	TIMEOUT : This is expressed as a number of clk_ref cycles and must be $\geq 2 \times \text{clk_ref_freq}/\text{min_clk_tst_freq}$	RW	0xff

CLOCKS: CLK_SYS_RESUS_STATUS Register

Offset: 0x7c

Table 238.
CLK_SYS_RESUS_STATUS
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	RESUSSED : Clock has been resuscitated, correct the error then send ctrl_clear=1	RO	0x0

CLOCKS: FC0_REF_KHZ Register

Offset: 0x80

表 235.
CLK_RTC_DIV 寄存器

位	描述	类型	复位值
31:8	INT : 除数的整数部分, 0 一除以 2^{16}	读写	0x000001
7:0	FRAC : 除数的小数部分	读写	0x00

CLOCKS: CLK_RTC_SELECTED 寄存器

偏移: 0x74

描述

指示防毛刺多路复用器（单热编码）当前选择的 SRC。

表 236.
CLK_RTC_SELECTED
寄存器

位	描述	类型	复位值
31:0	该切片不具备无毛刺多路复用器（仅包括 AUX_SRC 字段，不包括 SRC 字段），因此该寄存器固定为 0x1。	只读	0x00000001

CLOCKS: CLK_SYS_RESUS_CTRL 寄存器

偏移: 0x78

表 237.
CLK_SYS_RESUS_CTRL
寄存器

位	描述	类型	复位值
31:17	保留。	-	-
16	CLEAR : 在触发故障已纠正后清除复位	读写	0x0
15:13	保留。	-	-
12	FRCE : 强制复位, 仅限测试用途	读写	0x0
11:9	保留。	-	-
8	ENABLE : 启用复位	读写	0x0
7:0	TIMEOUT : 以clk_ref周期数表示, 且必须 $\geq 2 \times \text{clk_ref_freq} / \text{min_clk_tst_freq}$	读写	0xff

CLOCKS: CLK_SYS_RESUS_STATUS 寄存器

偏移: 0x7c

表 238.
CLK_SYS_RESUS_STATUS
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RESUSSED : 时钟已恢复, 纠正错误后发送 <i>ctrl_clear=1</i>	只读	0x0

CLOCKS: FC0_REF_KHZ 寄存器

偏移: 0x80

Table 239.
FC0_REF_KHZ Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19:0	Reference clock frequency in kHz	RW	0x00000

CLOCKS: FC0_MIN_KHZ Register

Offset: 0x84

Table 240.
FC0_MIN_KHZ Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24:0	Minimum pass frequency in kHz. This is optional. Set to 0 if you are not using the pass/fail flags	RW	0x0000000

CLOCKS: FC0_MAX_KHZ Register

Offset: 0x88

Table 241.
FC0_MAX_KHZ Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24:0	Maximum pass frequency in kHz. This is optional. Set to 0x1fffff if you are not using the pass/fail flags	RW	0x1fffff

CLOCKS: FC0_DELAY Register

Offset: 0x8c

Table 242. FC0_DELAY Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2:0	Delays the start of frequency counting to allow the mux to settle Delay is measured in multiples of the reference clock period	RW	0x1

CLOCKS: FC0_INTERVAL Register

Offset: 0x90

Table 243.
FC0_INTERVAL Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3:0	The test interval is 0.98us * 2**interval, but let's call it 1us * 2**interval The default gives a test interval of 250us	RW	0x8

CLOCKS: FC0_SRC Register

Offset: 0x94

Table 244. FC0_SRC Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	Clock sent to frequency counter, set to 0 when not required Writing to this register initiates the frequency count	RW	0x00
	Enumerated values:		
	0x00 → NULL		

表239
FC0_REF_KHZ寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:0	参考时钟频率，单位 kHz	读写	0x00000

CLOCKS: FC0_MIN_KHZ寄存器

偏移: 0x84

表240
FC0_MIN_KHZ
Z寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24:0	最小通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0。	读写	0x0000000

CLOCKS: FC0_MAX_KHZ寄存器

偏移: 0x88

表241。
FC0_MAX_KHZ
寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24:0	最大通过频率，单位 kHz。此项可选，不使用通过/失败标志时设置为0x1ffff fff。	读写	0x1fffffff

时钟：FC0_DELAY寄存器

偏移: 0x8c

表242. FC0_DELAY
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2:0	延迟频率计数启动，以使多路复用器稳定 延迟以参考时钟周期的整数倍计量	读写	0x1

时钟：FC0_INTERVAL寄存器

偏移: 0x90

表243。
FC0_INTERVAL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	测试间隔为0.98微秒 * 2**interval，但我们称之为1微秒 * 2**interval 默认测试间隔为250微秒	读写	0x8

时钟：FC0_SRC寄存器

偏移: 0x94

表244. FC0_SRC
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	传送至频率计数器的时钟，非必要时设置为0 写入此寄存器即可启动频率计数	读写	0x00
	枚举值：		
	0x00 → NULL		

Bits	Description	Type	Reset
	0x01 → PLL_SYS_CLKSRC_PRIMARY		
	0x02 → PLL_USB_CLKSRC_PRIMARY		
	0x03 → ROSC_CLKSRC		
	0x04 → ROSC_CLKSRC_PH		
	0x05 → XOSC_CLKSRC		
	0x06 → CLKSRC_GPIN0		
	0x07 → CLKSRC_GPIN1		
	0x08 → CLK_REF		
	0x09 → CLK_SYS		
	0x0a → CLK_PERI		
	0x0b → CLK_USB		
	0x0c → CLK_ADC		
	0x0d → CLK_RTC		

CLOCKS: FC0_STATUS Register

Offset: 0x98

Description

Frequency counter status

Table 245.
FC0_STATUS Register

Bits	Description	Type	Reset
31:29	Reserved.	-	-
28	DIED : Test clock stopped during test	RO	0x0
27:25	Reserved.	-	-
24	FAST : Test clock faster than expected, only valid when status_done=1	RO	0x0
23:21	Reserved.	-	-
20	SLOW : Test clock slower than expected, only valid when status_done=1	RO	0x0
19:17	Reserved.	-	-
16	FAIL : Test failed	RO	0x0
15:13	Reserved.	-	-
12	WAITING : Waiting for test clock to start	RO	0x0
11:9	Reserved.	-	-
8	RUNNING : Test running	RO	0x0
7:5	Reserved.	-	-
4	DONE : Test complete	RO	0x0
3:1	Reserved.	-	-
0	PASS : Test passed	RO	0x0

CLOCKS: FC0_RESULT Register

位	描述	类型	复位值
0x01	→ PLL_SYS_CLKSRC_PRIMARY		
0x02	→ PLL_USB_CLKSRC_PRIMARY		
0x03	→ ROSC_CLKSRC		
0x04	→ ROSC_CLKSRC_PH		
0x05	→ XOSC_CLKSRC		
0x06	→ CLKSRC_GPIN0		
0x07	→ CLKSRC_GPIN1		
0x08	→ CLK_REF		
0x09	→ CLK_SYS		
0x0a	→ CLK_PERI		
0x0b	→ CLK_USB		
0x0c	→ CLK_ADC		
0x0d	→ CLK_RTC		

CLOCKS: FC0_STATUS 寄存器

偏移: 0x98

说明

频率计数器状态

表 245。
FC0_STATUS 寄存器

位	描述	类型	复位值
31:29	保留。	-	-
28	DIED : 测试时钟在测试过程中停止	只读	0x0
27:25	保留。	-	-
24	FAST : 测试时钟快于预期, 仅当 status_done=1 时有效	只读	0x0
23:21	保留。	-	-
20	SLOW : 测试时钟慢于预期, 仅当 status_done=1 时有效	只读	0x0
19:17	保留。	-	-
16	FAIL : 测试失败	只读	0x0
15:13	保留。	-	-
12	等待中: 等待测试时钟启动	只读	0x0
11:9	保留。	-	-
8	运行中: 测试运行中	只读	0x0
7:5	保留。	-	-
4	完成: 测试完成	只读	0x0
3:1	保留。	-	-
0	通过: 测试通过	只读	0x0

时钟: FC0_RESULT 寄存器

Offset: 0x9c

Description

Result of frequency measurement, only valid when status_done=1

Table 246.
FC0_RESULT Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:5	KHZ	RO	0x0000000
4:0	FRAC	RO	0x00

CLOCKS: WAKE_EN0 Register

Offset: 0xa0

Description

enable clock in wake mode

Table 247. WAKE_EN0 Register

Bits	Description	Type	Reset
31	CLK_SYS_SRAM3	RW	0x1
30	CLK_SYS_SRAM2	RW	0x1
29	CLK_SYS_SRAM1	RW	0x1
28	CLK_SYS_SRAM0	RW	0x1
27	CLK_SYS_SPI1	RW	0x1
26	CLK_PERI_SPI1	RW	0x1
25	CLK_SYS_SPI0	RW	0x1
24	CLK_PERI_SPI0	RW	0x1
23	CLK_SYS_SIO	RW	0x1
22	CLK_SYS_RTC	RW	0x1
21	CLK_RTC_RTC	RW	0x1
20	CLK_SYS_ROSC	RW	0x1
19	CLK_SYS_ROM	RW	0x1
18	CLK_SYS_RESETS	RW	0x1
17	CLK_SYS_PWM	RW	0x1
16	CLK_SYS_PSM	RW	0x1
15	CLK_SYS_PLL_USB	RW	0x1
14	CLK_SYS_PLL_SYS	RW	0x1
13	CLK_SYS_PIO1	RW	0x1
12	CLK_SYS_PIO0	RW	0x1
11	CLK_SYS_PADS	RW	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	RW	0x1
9	CLK_SYS_JTAG	RW	0x1
8	CLK_SYS_IO	RW	0x1

偏移: 0x9c

描述

频率测量结果，仅当 status_done=1 时有效

表 246.
FC0_RESULT 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:5	KHZ	只读	0x00000000
4:0	FRAC	只读	0x00

时钟： WAKE_EN0 寄存器

偏移: 0xa0

描述

启用唤醒模式下的时钟

表247. WAKE_EN0寄存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	读写	0x1
30	CLK_SYS_SRAM2	读写	0x1
29	CLK_SYS_SRAM1	读写	0x1
28	CLK_SYS_SRAM0	读写	0x1
27	CLK_SYS_SPI1	读写	0x1
26	CLK_PERI_SPI1	读写	0x1
25	CLK_SYS_SPI0	读写	0x1
24	CLK_PERI_SPI0	读写	0x1
23	CLK_SYS_SIO	读写	0x1
22	CLK_SYS_RTC	读写	0x1
21	CLK_RTC_RTC	读写	0x1
20	CLK_SYS_ROSC	读写	0x1
19	CLK_SYS_ROM	读写	0x1
18	CLK_SYS_RESETS	读写	0x1
17	CLK_SYS_PWM	读写	0x1
16	CLK_SYS_PSM	读写	0x1
15	CLK_SYS_PLL_USB	读写	0x1
14	CLK_SYS_PLL_SYS	读写	0x1
13	CLK_SYS_PIO1	读写	0x1
12	CLK_SYS_PIO0	读写	0x1
11	CLK_SYS_PADS	读写	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	读写	0x1
9	CLK_SYS_JTAG	读写	0x1
8	CLK_SYS_IO	读写	0x1

Bits	Description	Type	Reset
7	CLK_SYS_I2C1	RW	0x1
6	CLK_SYS_I2C0	RW	0x1
5	CLK_SYS_DMA	RW	0x1
4	CLK_SYS_BUSFABRIC	RW	0x1
3	CLK_SYS_BUSCTRL	RW	0x1
2	CLK_SYS_ADC	RW	0x1
1	CLK_ADC_ADC	RW	0x1
0	CLK_SYS_CLOCKS	RW	0x1

CLOCKS: WAKE_EN1 Register

Offset: 0xa4

Description

enable clock in wake mode

Table 248. WAKE_EN1 Register

Bits	Description	Type	Reset
31:15	Reserved.	-	-
14	CLK_SYS_XOSC	RW	0x1
13	CLK_SYS_XIP	RW	0x1
12	CLK_SYS_WATCHDOG	RW	0x1
11	CLK_USB_USBCTRL	RW	0x1
10	CLK_SYS_USBCTRL	RW	0x1
9	CLK_SYS_UART1	RW	0x1
8	CLK_PERI_UART1	RW	0x1
7	CLK_SYS_UART0	RW	0x1
6	CLK_PERI_UART0	RW	0x1
5	CLK_SYS_TIMER	RW	0x1
4	CLK_SYS_TBMAN	RW	0x1
3	CLK_SYS_SYSINFO	RW	0x1
2	CLK_SYS_SYSCFG	RW	0x1
1	CLK_SYS_SRAM5	RW	0x1
0	CLK_SYS_SRAM4	RW	0x1

CLOCKS: SLEEP_EN0 Register

Offset: 0xa8

Description

enable clock in sleep mode

Table 249. SLEEP_EN0 Register

Bits	Description	Type	Reset
31	CLK_SYS_SRAM3	RW	0x1

位	描述	类型	复位值
7	CLK_SYS_I2C1	读写	0x1
6	CLK_SYS_I2C0	读写	0x1
5	CLK_SYS_DMA	读写	0x1
4	CLK_SYS_BUSFABRIC	读写	0x1
3	CLK_SYS_BUSCTRL	读写	0x1
2	CLK_SYS_ADC	读写	0x1
1	CLK_ADC_ADC	读写	0x1
0	CLK_SYS_CLOCKS	读写	0x1

时钟：WAKE_EN1 寄存器

偏移: 0xa4

描述

启用唤醒模式下的时钟

表248. WAKE_EN1寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	读写	0x1
13	CLK_SYS_XIP	读写	0x1
12	CLK_SYS_WATCHDOG	读写	0x1
11	CLK_USB_USBCTRL	读写	0x1
10	CLK_SYS_USBCTRL	读写	0x1
9	CLK_SYS_UART1	读写	0x1
8	CLK_PERI_UART1	读写	0x1
7	CLK_SYS_UART0	读写	0x1
6	CLK_PERI_UART0	读写	0x1
5	CLK_SYS_TIMER	读写	0x1
4	CLK_SYS_TBMAN	读写	0x1
3	CLK_SYS_SYSINFO	读写	0x1
2	CLK_SYS_SYSCFG	读写	0x1
1	CLK_SYS_SRAM5	读写	0x1
0	CLK_SYS_SRAM4	读写	0x1

时钟：SLEEP_EN0 寄存器

偏移: 0xa8

描述

启用睡眠模式下的时钟

表249. SLEEP_EN0寄存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	读写	0x1

Bits	Description	Type	Reset
30	CLK_SYS_SRAM2	RW	0x1
29	CLK_SYS_SRAM1	RW	0x1
28	CLK_SYS_SRAM0	RW	0x1
27	CLK_SYS_SPI1	RW	0x1
26	CLK_PERI_SPI1	RW	0x1
25	CLK_SYS_SPI0	RW	0x1
24	CLK_PERI_SPI0	RW	0x1
23	CLK_SYS_SIO	RW	0x1
22	CLK_SYS_RTC	RW	0x1
21	CLK_RTC_RTC	RW	0x1
20	CLK_SYS_ROSC	RW	0x1
19	CLK_SYS_ROM	RW	0x1
18	CLK_SYS_RESETS	RW	0x1
17	CLK_SYS_PWM	RW	0x1
16	CLK_SYS_PSM	RW	0x1
15	CLK_SYS_PLL_USB	RW	0x1
14	CLK_SYS_PLL_SYS	RW	0x1
13	CLK_SYS_PIO1	RW	0x1
12	CLK_SYS_PIO0	RW	0x1
11	CLK_SYS_PADS	RW	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	RW	0x1
9	CLK_SYS_JTAG	RW	0x1
8	CLK_SYS_IO	RW	0x1
7	CLK_SYS_I2C1	RW	0x1
6	CLK_SYS_I2C0	RW	0x1
5	CLK_SYS_DMA	RW	0x1
4	CLK_SYS_BUSFABRIC	RW	0x1
3	CLK_SYS_BUSCTRL	RW	0x1
2	CLK_SYS_ADC	RW	0x1
1	CLK_ADC_ADC	RW	0x1
0	CLK_SYS_CLOCKS	RW	0x1

CLOCKS: SLEEP_EN1 Register

Offset: 0xac

Description

enable clock in sleep mode

位	描述	类型	复位值
30	CLK_SYS_SRAM2	读写	0x1
29	CLK_SYS_SRAM1	读写	0x1
28	CLK_SYS_SRAM0	读写	0x1
27	CLK_SYS_SPI1	读写	0x1
26	CLK_PERI_SPI1	读写	0x1
25	CLK_SYS_SPI0	读写	0x1
24	CLK_PERI_SPI0	读写	0x1
23	CLK_SYS_SIO	读写	0x1
22	CLK_SYS_RTC	读写	0x1
21	CLK_RTC_RTC	读写	0x1
20	CLK_SYS_ROSC	读写	0x1
19	CLK_SYS_ROM	读写	0x1
18	CLK_SYS_RESETS	读写	0x1
17	CLK_SYS_PWM	读写	0x1
16	CLK_SYS_PSM	读写	0x1
15	CLK_SYS_PLL_USB	读写	0x1
14	CLK_SYS_PLL_SYS	读写	0x1
13	CLK_SYS_PIO1	读写	0x1
12	CLK_SYS_PIO0	读写	0x1
11	CLK_SYS_PADS	读写	0x1
10	CLK_SYS_VREG_AND_CHIP_RESET	读写	0x1
9	CLK_SYS_JTAG	读写	0x1
8	CLK_SYS_IO	读写	0x1
7	CLK_SYS_I2C1	读写	0x1
6	CLK_SYS_I2C0	读写	0x1
5	CLK_SYS_DMA	读写	0x1
4	CLK_SYS_BUSFABRIC	读写	0x1
3	CLK_SYS_BUSCTRL	读写	0x1
2	CLK_SYS_ADC	读写	0x1
1	CLK_ADC_ADC	读写	0x1
0	CLK_SYS_CLOCKS	读写	0x1

CLOCKS: SLEEP_EN1 寄存器

偏移: 0xac

描述

启用睡眠模式下的时钟

Table 250. SLEEP_EN1 Register

Bits	Description	Type	Reset
31:15	Reserved.	-	-
14	CLK_SYS_XOSC	RW	0x1
13	CLK_SYS_XIP	RW	0x1
12	CLK_SYS_WATCHDOG	RW	0x1
11	CLK_USB_USBCTRL	RW	0x1
10	CLK_SYS_USBCTRL	RW	0x1
9	CLK_SYS_UART1	RW	0x1
8	CLK_PERI_UART1	RW	0x1
7	CLK_SYS_UART0	RW	0x1
6	CLK_PERI_UART0	RW	0x1
5	CLK_SYS_TIMER	RW	0x1
4	CLK_SYS_TBMAN	RW	0x1
3	CLK_SYS_SYSINFO	RW	0x1
2	CLK_SYS_SYSCFG	RW	0x1
1	CLK_SYS_SRAM5	RW	0x1
0	CLK_SYS_SRAM4	RW	0x1

CLOCKS: ENABLED0 Register

Offset: 0xb0

Description

indicates the state of the clock enable

Table 251. ENABLED0 Register

Bits	Description	Type	Reset
31	CLK_SYS_SRAM3	RO	0x0
30	CLK_SYS_SRAM2	RO	0x0
29	CLK_SYS_SRAM1	RO	0x0
28	CLK_SYS_SRAM0	RO	0x0
27	CLK_SYS_SPI1	RO	0x0
26	CLK_PERI_SPI1	RO	0x0
25	CLK_SYS_SPI0	RO	0x0
24	CLK_PERI_SPI0	RO	0x0
23	CLK_SYS_SIO	RO	0x0
22	CLK_SYS_RTC	RO	0x0
21	CLK_RTC_RTC	RO	0x0
20	CLK_SYS_ROSC	RO	0x0
19	CLK_SYS_ROM	RO	0x0
18	CLK_SYS_RESETS	RO	0x0

表250. SLEEP_EN1寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	读写	0x1
13	CLK_SYS_XIP	读写	0x1
12	CLK_SYS_WATCHDOG	读写	0x1
11	CLK_USB_USBCTRL	读写	0x1
10	CLK_SYS_USBCTRL	读写	0x1
9	CLK_SYS_UART1	读写	0x1
8	CLK_PERI_UART1	读写	0x1
7	CLK_SYS_UART0	读写	0x1
6	CLK_PERI_UART0	读写	0x1
5	CLK_SYS_TIMER	读写	0x1
4	CLK_SYS_TBMAN	读写	0x1
3	CLK_SYS_SYSINFO	读写	0x1
2	CLK_SYS_SYSCFG	读写	0x1
1	CLK_SYS_SRAM5	读写	0x1
0	CLK_SYS_SRAM4	读写	0x1

CLOCKS: ENABLEDO 寄存器

偏移: 0xb0

描述

指示时钟使能状态

表 251. ENABLEDO 寄存器

位	描述	类型	复位值
31	CLK_SYS_SRAM3	只读	0x0
30	CLK_SYS_SRAM2	只读	0x0
29	CLK_SYS_SRAM1	只读	0x0
28	CLK_SYS_SRAM0	只读	0x0
27	CLK_SYS_SPI1	只读	0x0
26	CLK_PERI_SPI1	只读	0x0
25	CLK_SYS_SPI0	只读	0x0
24	CLK_PERI_SPI0	只读	0x0
23	CLK_SYS_SIO	只读	0x0
22	CLK_SYS_RTC	只读	0x0
21	CLK_RTC_RTC	只读	0x0
20	CLK_SYS_ROSC	只读	0x0
19	CLK_SYS_ROM	只读	0x0
18	CLK_SYS_RESETS	只读	0x0

Bits	Description	Type	Reset
17	CLK_SYS_PWM	RO	0x0
16	CLK_SYS_PSM	RO	0x0
15	CLK_SYS_PLL_USB	RO	0x0
14	CLK_SYS_PLL_SYS	RO	0x0
13	CLK_SYS_PIO1	RO	0x0
12	CLK_SYS_PIO0	RO	0x0
11	CLK_SYS_PADS	RO	0x0
10	CLK_SYS_VREG_AND_CHIP_RESET	RO	0x0
9	CLK_SYS_JTAG	RO	0x0
8	CLK_SYS_IO	RO	0x0
7	CLK_SYS_I2C1	RO	0x0
6	CLK_SYS_I2C0	RO	0x0
5	CLK_SYS_DMA	RO	0x0
4	CLK_SYS_BUSFABRIC	RO	0x0
3	CLK_SYS_BUSCTRL	RO	0x0
2	CLK_SYS_ADC	RO	0x0
1	CLK_ADC_ADC	RO	0x0
0	CLK_SYS_CLOCKS	RO	0x0

CLOCKS: ENABLED1 Register

Offset: 0xb4

Description

indicates the state of the clock enable

Table 252. ENABLED1 Register

Bits	Description	Type	Reset
31:15	Reserved.	-	-
14	CLK_SYS_XOSC	RO	0x0
13	CLK_SYS_XIP	RO	0x0
12	CLK_SYS_WATCHDOG	RO	0x0
11	CLK_USB_USBCTRL	RO	0x0
10	CLK_SYS_USBCTRL	RO	0x0
9	CLK_SYS_UART1	RO	0x0
8	CLK_PERI_UART1	RO	0x0
7	CLK_SYS_UART0	RO	0x0
6	CLK_PERI_UART0	RO	0x0
5	CLK_SYS_TIMER	RO	0x0
4	CLK_SYS_TBMAN	RO	0x0

位	描述	类型	复位值
17	CLK_SYS_PWM	只读	0x0
16	CLK_SYS_PSM	只读	0x0
15	CLK_SYS_PLL_USB	只读	0x0
14	CLK_SYS_PLL_SYS	只读	0x0
13	CLK_SYS_PIO1	只读	0x0
12	CLK_SYS_PIO0	只读	0x0
11	CLK_SYS_PADS	只读	0x0
10	CLK_SYS_VREG_AND_CHIP_RESET	只读	0x0
9	CLK_SYS_JTAG	只读	0x0
8	CLK_SYS_IO	只读	0x0
7	CLK_SYS_I2C1	只读	0x0
6	CLK_SYS_I2C0	只读	0x0
5	CLK_SYS_DMA	只读	0x0
4	CLK_SYS_BUSFABRIC	只读	0x0
3	CLK_SYS_BUSCTRL	只读	0x0
2	CLK_SYS_ADC	只读	0x0
1	CLK_ADC_ADC	只读	0x0
0	CLK_SYS_CLOCKS	只读	0x0

CLOCKS: ENABLED1 寄存器

偏移: 0xb4

描述

指示时钟使能状态

表 252. ENABLED1
寄存器

位	描述	类型	复位值
31:15	保留。	-	-
14	CLK_SYS_XOSC	只读	0x0
13	CLK_SYS_XIP	只读	0x0
12	CLK_SYS_WATCHDOG	只读	0x0
11	CLK_USB_USBCTRL	只读	0x0
10	CLK_SYS_USBCTRL	只读	0x0
9	CLK_SYS_UART1	只读	0x0
8	CLK_PERI_UART1	只读	0x0
7	CLK_SYS_UART0	只读	0x0
6	CLK_PERI_UART0	只读	0x0
5	CLK_SYS_TIMER	只读	0x0
4	CLK_SYS_TBMAN	只读	0x0

Bits	Description	Type	Reset
3	CLK_SYS_SYSINFO	RO	0x0
2	CLK_SYS_SYSCFG	RO	0x0
1	CLK_SYS_SRAM5	RO	0x0
0	CLK_SYS_SRAM4	RO	0x0

CLOCKS: INTR Register

Offset: 0xb8

Description

Raw Interrupts

Table 253. INTR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLK_SYS_RESUS	RO	0x0

CLOCKS: INTE Register

Offset: 0xbc

Description

Interrupt Enable

Table 254. INTE Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLK_SYS_RESUS	RW	0x0

CLOCKS: INTF Register

Offset: 0xc0

Description

Interrupt Force

Table 255. INTF Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLK_SYS_RESUS	RW	0x0

CLOCKS: INTS Register

Offset: 0xc4

Description

Interrupt status after masking & forcing

位	描述	类型	复位值
3	CLK_SYS_SYSINFO	只读	0x0
2	CLK_SYS_SYSCFG	只读	0x0
1	CLK_SYS_SRAM5	只读	0x0
0	CLK_SYS_SRAM4	只读	0x0

CLOCKS: INTR 寄存器

偏移: 0xb8

描述

原始中断

表 253. INTR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	只读	0x0

CLOCKS: INTF 寄存器

偏移: 0xbc

描述

中断使能

表 254. INTF
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	读写	0x0

CLOCKS: INTS 寄存器

偏移: 0xc0

描述

中断强制

表 255. INTS
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	读写	0x0

CLOCKS: INTS 寄存器

偏移: 0xc4

描述

掩码及强制后的中断状态

Table 256. INTS Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLK_SYS_RESUS	RO	0x0

2.16. Crystal Oscillator (XOSC)

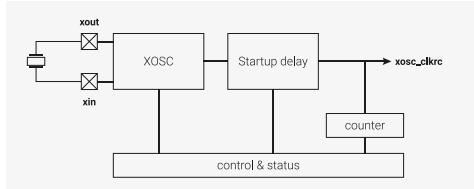
2.16.1. Overview

The Crystal Oscillator (XOSC) uses an external crystal to produce an accurate reference clock. The RP2040 supports 1MHz to 15MHz crystals and the RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#)) uses a 12MHz crystal. The reference clock is distributed to the PLLs, which can be used to multiply the XOSC frequency to provide accurate high speed clocks. For example, they can generate a 48MHz clock which meets the frequency accuracy requirement of the USB interface and a 133MHz maximum speed system clock. The XOSC clock is also a clock source for the clock generators, so can be used directly if required.

If the user already has an accurate clock source then it is possible to drive an external clock directly into XIN (aka XI), and disable the oscillator circuit. In this mode XIN can be driven at up to 50MHz.

If the user wants to use the XOSC clock outside the RP2040 then it must be routed out to a GPIO via a clk_gout clock generator. It is not recommended to take it directly from XIN (aka XI) or XOUT (aka XO).

Figure 33. XOSC overview



2.16.1.1. Recommended Crystals

For the best performance and stability across typical operating temperature ranges, it is recommended to use the Abraccon ABM8-272-T3. You can source the ABM8-272-T3 directly from Abraccon or from an authorised reseller. The Abraccon ABM8-272-T3 has the following specifications:

Table 257. Key Crystal Specifications.

Parameters	Minimum	Typical	Maximum	Units	Notes
Center Frequency	12.000	12.000	12.000	MHz	
Operation Mode	Fundamental-AT	Fundamental-AT	Fundamental-AT		
Operating Temperature	-40		+85	°C	
Storage Temperature	-55		+125	°C	
Frequency Tolerance (25°C)	-30		+30	ppm	
Frequency Stability (25°C)	-30		+30	ppm	
Equivalent Series Resistance (R1)			50	Ω	
Shunt Capacitance (C0)			3.0	pF	
Load Capacitance (CL)	10	10	10	pF	
Drive Level		10	200	μW	
Aging	-5		+5	ppm	@25±3°C, 1st year

表 256. INT3
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLK_SYS_RESUS	只读	0x0

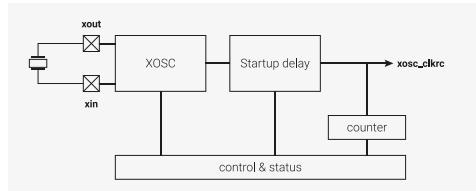
2.16. 晶体振荡器 (XOSC)

2.16.1. 概述

晶振 (XOSC) 采用外部晶体以提供精确的参考时钟。RP2040支持1MHz至15MHz晶体，RP2040参考设计（见《基于RP2040的硬件设计》中的最小设计示例）采用12MHz晶体。参考时钟分配给PLL，用于倍频XOSC频率，提供精确的高速时钟。例如，PLL可生成满足USB接口频率精度要求的48MHz时钟，以及最高133MHz的系统时钟。XOSC时钟也是时钟发生器的时钟源，因此在需要时可直接使用。

如用户已有精确时钟源，可直接将外部时钟输入XIN（即XI）引脚，且可禁用振荡电路。在该模式下，XIN的驱动频率最高可达50MHz。

如果用户希望在RP2040之外使用XOSC时钟，则必须通过clk_gpout时钟发生器将其路由至GPIO端口。不建议直接从XIN（又称XI）或XOUT（又称XO）获取信号。

图33. XOSC
概述

2.16.1.1 推荐晶体

为在典型工作温度范围内实现最佳性能与稳定性，建议使用Abracon ABM8-272-T3。您可直接向Abracon或其授权经销商采购该型号。Abracon ABM8-272-T3具有以下规格：

表257。关键晶体
规格

参数	最小值	典型值	最大值	单位	备注
中心频率	12.000	12.000	12.000	MHz	
工作模式	Fundamental-AT	Fundamental-AT	Fundamental-AT		
工作温度	-40		+85	°C	
存储温度	-55		+125	°C	
频率容差 (25°C)	-30		+30	ppm	
频率稳定性 (25°C)	-30		+30	ppm	
等效串联电阻 (R1)			50	Ω	
并联电容 (C0)			3.0	pF	
负载电容 (CL)	10	10	10	pF	
驱动功率		10	200	μW	
老化	-5		+5	ppm	@25±3°C, 第一年

Parameters	Minimum	Typical	Maximum	Units	Notes
Insulation Resistance	500			MΩ	@100Vdc±15V

Even if you use a crystal with similar specifications, you will need to test the circuit over a range of temperatures to ensure stability.

The crystal oscillator is powered from the VDDIO voltage. As a result, the Abracan crystal and that particular damping resistor are tuned for 3.3V operation. If you use a different IO voltage, you will need to re-tune.

Any changes to crystal parameters risk instability across any components connected to the crystal circuit.

If you can't source the recommended crystal directly from Abracan or a reseller, contact applications@raspberrypi.com.

Raspberry Pi Pico has been specifically tuned for the specifications of the Abracan ABM8-272-T3 crystal. For an example of how to use a crystal with RP2040, see the Raspberry Pi Pico board schematic in [Appendix B of the Raspberry Pi Pico Datasheet](#) and the [Raspberry Pi Pico design files](#).

2.16.2. Usage

The XOSC is disabled on chip startup and the RP2040 boots using the Ring Oscillator (ROSC). To start the XOSC, the programmer must set the CTRL_ENABLE register. The XOSC is not immediately usable because it takes time for the oscillations to build to sufficient amplitude. This time will be dependent on the chosen crystal but will be of the order of a few milliseconds. The XOSC incorporates a timer controlled by the STARTUP_DELAY register for automatically managing this and setting a flag (STATUS_STABLE) when the XOSC clock is usable.

2.16.3. Startup Delay

The STARTUP_DELAY register specifies how many clock cycles must be seen from the crystal before it can be used. This is specified in multiples of 256. The SDK `xosc_init` function sets this value. The 1ms default is sufficient for the RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#)) which runs the XOSC at 12MHz. When the timer expires, the STATUS_STABLE flag will be set to indicate the XOSC output can be used.

Before starting the XOSC the programmer must ensure the STARTUP_DELAY register is correctly configured. The required value can be calculated by:

$$(f_{Crystal} \times t_{Stable}) \div 256$$

So with a 12MHz crystal and a 1ms wait time, the calculation is:

$$(12 \times 10^6 \cdot 1 \times 10^{-3}) \div 256 \approx 47$$

NOTE

The value is rounded up to the nearest integer so the wait time will be just over 1ms

2.16.4. XOSC Counter

The COUNT register provides a method of managing short software delays. Writing a value to the COUNT register automatically triggers it to start counting down to zero at the XOSC frequency. The programmer then simply polls the register until it reaches zero. This is preferable to using NOPs in software loops because it is independent of the core clock frequency, the compiler and the execution time of the compiled code.

参数	最小值	典型值	最大值	单位	备注
绝缘电阻	500			MΩ	@100Vdc±15V

即使使用规格相似的晶体振荡器，仍需在不同温度范围内测试电路以确保稳定性。

晶体振荡器由 VDDIO 电压供电。因此，Abracon 晶体及其特定阻尼电阻已经针对 3.3V 电压进行了调试。如果使用不同的 IO 电压，则需要重新调试。

对晶体参数的任何更改都可能导致连接于晶体电路的任何组件出现不稳定。

如果无法直接从 Abracon 或其经销商处采购推荐的晶体，请联系 applications@raspberrypi.com。

Raspberry Pi Pico 已专门针对 Abracon ABM8-272-T3 晶体的规格进行调试。有关在 RP2040 中使用晶体的示例，请参见 Raspberry Pi Pico 数据手册附录 B 中的 Raspberry Pi Pico 板原理图及设计文件。

2.16.2. 使用方法

XOSC 在芯片启动时处于禁用状态，RP2040 使用环形振荡器（ROSC）启动。要启动 XOSC，程序员必须设置 CTRL_ENABLE 寄存器。XOSC不可立即使用，因为振荡需要时间达到足够的幅度。该时间取决于所选晶体，但通常为几毫秒的量级。XOSC包含由STARTUP_DELAY寄存器控制的定时器，用于自动管理该过程，并在XOSC时钟可用时设置状态标志（STATUS_STABLE）。

2.16.3. 启动延迟

STARTUP_DELAY寄存器指定晶体必须检测到的时钟周期数，方可使用。

该数值以256的整数倍表示。该值由SDK中的 `xosc_init` 函数设置。1毫秒的默认值适用于RP2040参考设计（详见RP2040 硬件设计中的最小设计示例），该设计以12MHz频率运行XOSC。定时器到期后，STATUS_STABLE标志将被置位，指示 XOSC输出可用。

启动XOSC之前，程序员必须确保STARTUP_DELAY寄存器配置正确。所需的数值可通过以下公式计算：

$$(f_{Crystal} \times t_{Stable}) \div 256$$

因此，采用12MHz晶振和1ms等待时间，计算结果如下：

$$(12 \times 10^6 \cdot 1 \times 10^{-3}) \div 256 \approx 47$$

注意

该值向上取整至最接近的整数，因此实际等待时间略超过1ms。

2.16.4. XOSC 计数器

COUNT寄存器提供了一种管理短时软件延迟的方法。向COUNT寄存器写入数值后，将自动以XOSC频率开始倒计时至零。程序员只需轮询该寄存器，直到其值达到零。此方法优于在软件循环中使用NOP指令，因为它不依赖核心时钟频率、编译器或编译后代码的执行时间。

2.16.5. DORMANT mode

In DORMANT mode (see [Section 2.11.3](#)) all of the on-chip clocks can be paused to save power. This is particularly useful in battery-powered applications. The RP2040 is woken from DORMANT mode by an interrupt either from an external event such as an edge on a GPIO pin or from the on-chip RTC. This must be configured before entering DORMANT mode. If the RTC is being used to trigger wake-up then it must be clocked from an external source. To enter DORMANT mode the programmer must then switch all internal clocks to be driven from XOSC or ROSC and stop the PLLs. Then a specific 32-bit value must be written to the DORMANT register in the chosen oscillator (XOSC or ROSC) to stop it oscillating. When exiting DORMANT mode the chosen oscillator will restart. If XOSC is chosen then the frequency will be more precise but the restart time is longer due to the startup delay (>1ms on the RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#))). If ROSC is chosen then the frequency is less precise but the start-up time is very short (approximately 1μs).

NOTE

The PLLs must be stopped before entering DORMANT mode

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c Lines 56 - 63

```
56 void xosc_dormant(void) {
57     // WARNING: This stops the xosc until woken up by an irq
58     xosc_hw->dormant = XOSC_DORMANT_VALUE_DORMANT;
59     // Wait for it to become stable once woken up
60     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
61         tight_loop_contents();
62     }
63 }
```

WARNING

If no IRQ is configured before going into DORMANT mode the XOSC or ROSC will never restart.

See [Section 2.11.5.2](#) for a complete example of DORMANT mode using the XOSC.

2.16.6. Programmer's Model

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/xosc.h Lines 27 - 59

```
27 typedef struct {
28     _REG_(XOSC_CTRL_OFFSET) // XOSC_CTRL
29     // Crystal Oscillator Control
30     // 0x00ffff00 [23:12] ENABLE      (-) On power-up this field is initialised to DISABLE and
31     // the...
32     // 0x00000fff [11:0] FREQ_RANGE   (-) Frequency range
33     io_rw_32 ctrl;
34
35     _REG_(XOSC_STATUS_OFFSET) // XOSC_STATUS
36     // Crystal Oscillator Status
37     // 0x80000000 [31] STABLE       (0) Oscillator is running and stable
38     // 0x01000000 [24] BADWRITE    (0) An invalid value has been written to CTRL_ENABLE
39     // or...
40     // 0x00001000 [12] ENABLED     (-) Oscillator is enabled but not necessarily running
41     // and...
42     // 0x00000003 [1:0] FREQ_RANGE (-) The current frequency range setting, always reads 0
43     io_rw_32 status;
44 }
```

2.16.5. 休眠模式

在休眠模式（详见第2.11.3节）下，所有片上时钟均可暂停以节省功耗。此功能在电池供电的应用中尤为重要。RP2040通过外部事件（如GPIO引脚的边沿变化）或片上RTC的中断唤醒休眠模式。此配置须在进入休眠模式前完成。如果实时钟（RTC）用于触发唤醒，则必须由外部时钟源提供时钟。要进入休眠模式，程序员必须将所有内部时钟切换为由XOSC或ROSC驱动，并停止PLL。随后，必须向所选振荡器（XOSC或ROSC）的DORMANT寄存器写入特定的32位数值以停止振荡。退出休眠模式时，所选振荡器将重新启动。若选择XOSC，则频率更为精准，但因启动延迟，重启时间较长（RP2040参考设计中超过1ms，详见《RP2040硬件设计》中的最简设计示例）。若选择ROSC，频率较不精准，但启动时间极短（约1μs）。

注意

进入休眠模式前必须停止PLL。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c 第56至63行

```
56 void xosc_dormant(void) {
57     // 警告：此操作将停止xosc，直至通过irq唤醒
58     xosc_hw->dormant = XOSC_DORMANT_VALUE_DORMANT;
59     // 唤醒后，等待其稳定
60     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
61         tight_loop_contents();
62     }
63 }
```

警告

若在进入休眠模式前未配置IRQ，则XOSC或ROSC将无法重新启动。

有关使用XOSC休眠模式的完整示例，请参见第2.11.5.2节。

2.16.6. 程程序员模型

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/xosc.h 第27至59行

```
27 typedef struct {
28     _REG_(XOSC_CTRL_OFFSET) // XOSC_CTRL
29     // 晶体振荡器控制
30     // 0x00ffff000 [23:12] ENABLE      (-) 上电时，该字段初始化为 DISABLE，并且
31     // 该...
32     // 0x00000fff [11:0]   FREQ_RANGE   (-) 频率范围
33     io_rw_32 ctrl;
34
35     _REG_(XOSC_STATUS_OFFSET) // XOSC_STATUS
36     // 晶体振荡器状态
37     // 0x80000000 [31]      STABLE       (0) 振荡器正在运行且处于稳定状态
38 // 0x01000000 [24] BADWRITE (0) 向 CTRL_ENABLE 或...
39
40     // 0x00001000 [12]      ENABLED      (-) 振荡器已启用，但不一定正在运行
41     // 并且...
42     // 0x00000003 [1:0]      FREQ_RANGE   (-) 当前频率范围设置，始终读取为 0
43     io_rw_32 status;
44 }
```

```

42     _REG_(XOSC_DORMANT_OFFSET) // XOSC_DORMANT
43     // Crystal Oscillator pause control
44     // 0xffffffff [31:0] DORMANT      (-) This is used to save power by pausing the XOSC +
45     io_rw_32 dormant;
46
47     _REG_(XOSC_STARTUP_OFFSET) // XOSC_STARTUP
48     // Controls the startup delay
49     // 0x00100000 [20]    X4          (-) Multiplies the startup_delay by 4
50     // 0x00003fff [13:0]  DELAY       (-) in multiples of 256*xtal_period
51     io_rw_32 startup;
52
53     uint32_t _pad0[3];
54
55     _REG_(XOSC_COUNT_OFFSET) // XOSC_COUNT
56     // A down counter running at the XOSC frequency which counts to zero and stops.
57     // 0x000000ff [7:0]   COUNT      (0x00)
58     io_rw_32 count;
59 } xosc_hw_t;

```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c Lines 29 - 43

```

29 void xosc_init(void) {
30     // Assumes 1-15 MHz input, checked above.
31     xosc_hw->ctrl = XOSC_CTRL_FREQ_RANGE_VALUE_1_15MHZ;
32
33     // Set xosc startup delay
34     xosc_hw->startup = STARTUP_DELAY;
35
36     // Set the enable bit now that we have set freq range and startup delay
37     hw_set_bits(&xosc_hw->ctrl, XOSC_CTRL_ENABLE_VALUE_ENABLE << XOSC_CTRL_ENABLE_LSB);
38
39     // Wait for XOSC to be stable
40     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
41         tight_loop_contents();
42     }
43 }

```

2.16.7. List of Registers

The XOSC registers start at a base address of **0x40024000** (defined as **XOSC_BASE** in SDK).

Table 258. List of XOSC registers

Offset	Name	Info
0x00	CTRL	Crystal Oscillator Control
0x04	STATUS	Crystal Oscillator Status
0x08	DORMANT	Crystal Oscillator pause control
0x0c	STARTUP	Controls the startup delay
0x1c	COUNT	A down counter running at the XOSC frequency which counts to zero and stops.

XOSC: CTRL Register

Offset: 0x00

```

42     _REG_(XOSC_DORMANT_OFFSET) // XOSC_DORMANT
43     // 晶体振荡器暂停控制
44     // 0xffffffff [31:0] DORMANT      (-) 用于通过暂停XOSC以节省功耗 +
45     io_rw_32 dormant;
46
47     _REG_(XOSC_STARTUP_OFFSET) // XOSC_STARTUP
48     // 控制启动延迟
49     // 0x00100000 [20]    X4          (-) 将startup_delay乘以4
50     // 0x00003fff [13:0]  DELAY       (-) 以256*xtal_period为单位的倍数
51     io_rw_32 startup;
52
53     uint32_t _pad0[3];
54
55     _REG_(XOSC_COUNT_OFFSET) // XOSC_COUNT
56     // 以XOSC频率运行的递减计数器，计数至零后停止。
57     // 0x000000ff [7:0]    COUNT      (0x00)
58     io_rw_32 count;
59 } xosc_hw_t;

```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_xosc/xosc.c 第29至43行

```

29 void xosc_init(void) {
30     // 假设输入频率为1-15 MHz，已在前文确认。
31     xosc_hw->ctrl = XOSC_CTRL_FREQ_RANGE_VALUE_1_15MHZ;
32
33     // 设置xosc启动延迟
34     xosc_hw->startup = STARTUP_DELAY;
35
36     // 设置使能位，频率范围及启动延迟已配置完成
37     hw_set_bits(&xosc_hw->ctrl, XOSC_CTRL_ENABLE_VALUE_ENABLE << XOSC_CTRL_ENABLE_LSB);
38
39     // 等待XOSC稳定
40     while(!(xosc_hw->status & XOSC_STATUS_STABLE_BITS)) {
41         tight_loop_contents();
42     }
43 }

```

2.16.7. 寄存器列表

XOSC寄存器起始基址为 **0x40024000**（在SDK中定义为XOSC_BASE）。

表258.
XOSC寄存器列表

偏移量	名称	说明
0x00	CTRL	晶体振荡器控制
0x04	状态	晶体振荡器状态
0x08	休眠	晶体振荡器暂停控制
0x0c	启动	控制启动延迟
0x1c	计数	一个以XOSC频率运行的递减计数器，计数至零后停止。

XOSC: CTRL寄存器

偏移: 0x00

Description

Crystal Oscillator Control

Table 259. CTRL Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:12	ENABLE: On power-up this field is initialised to DISABLE and the chip runs from the ROSC. If the chip has subsequently been programmed to run from the XOSC then setting this field to DISABLE may lock-up the chip. If this is a concern then run the clk_ref from the ROSC and enable the clk_sys RESUS feature. The 12-bit code is intended to give some protection against accidental writes. An invalid setting will enable the oscillator.	RW	-
	Enumerated values: 0xd1e → DISABLE		
	0xfb → ENABLE		
11:0	FREQ_RANGE: Frequency range. This resets to 0xAA0 and cannot be changed.	RW	-
	Enumerated values: 0xaa0 → 1_15MHZ		
	0xaa1 → RESERVED_1		
	0xaa2 → RESERVED_2		
	0xaa3 → RESERVED_3		

XOSC: STATUS Register

Offset: 0x04

Description

Crystal Oscillator Status

Table 260. STATUS Register

Bits	Description	Type	Reset
31	STABLE: Oscillator is running and stable	RO	0x0
30:25	Reserved.	-	-
24	BADWRITE: An invalid value has been written to CTRL_ENABLE or CTRL_FREQ_RANGE or DORMANT	WC	0x0
23:13	Reserved.	-	-
12	ENABLED: Oscillator is enabled but not necessarily running and stable, resets to 0	RO	-
11:2	Reserved.	-	-
1:0	FREQ_RANGE: The current frequency range setting, always reads 0	RO	-
	Enumerated values: 0x0 → 1_15MHZ		
	0x1 → RESERVED_1		
	0x2 → RESERVED_2		
	0x3 → RESERVED_3		

描述

晶体振荡器控制

表259.
CTRL寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	启用: 上电时, 该字段初始化为禁用, 芯片运行于ROSC。 若芯片后来被配置为由XOSC供电, 则将此字段设为禁用可能导致芯片锁死。如有此风险, 建议将clk_ref运行于ROSC, 并启用clk_sys的RESUS功能。 该12位代码旨在防止意外写入。 无效的设置将启用振荡器。	读写	-
	枚举值:		
	0xd1e → DISABLE		
	0xfb → ENABLE		
11:0	FREQ_RANGE: 频率范围。此值重置为0xAA0, 且不可更改。	读写	-
	枚举值:		
	0xaa0 → 1_15MHZ		
	0xaa1 → RESERVED_1		
	0xaa2 → RESERVED_2		
	0xaa3 → RESERVED_3		

XOSC: STATUS寄存器

偏移: 0x04

描述

晶体振荡器状态

表260. STATUS
寄存器

位	描述	类型	复位值
31	STABLE: 振荡器正在运行且稳定	只读	0x0
30:25	保留。	-	-
24	BADWRITE: 已向CTRL_ENABLE、CTRL_FREQ_RANGE或DORMANT写入无效值	WC	0x0
23:13	保留。	-	-
12	ENABLED: 振荡器已启用, 但不一定正在运行且稳定, 复位值为0	只读	-
11:2	保留。	-	-
1:0	FREQ_RANGE: 当前频率范围设置, 始终读取为0	只读	-
	枚举值:		
	0x0 → 1_15MHZ		
	0x1 → RESERVED_1		
	0x2 → RESERVED_2		
	0x3 → RESERVED_3		

XOSC: DORMANT Register

Offset: 0x08

Description

Crystal Oscillator pause control

Table 261. DORMANT Register

Bits	Description	Type	Reset
31:0	This is used to save power by pausing the XOSC On power-up this field is initialised to WAKE An invalid write will also select WAKE WARNING: stop the PLLs before selecting dormant mode WARNING: setup the irq before selecting dormant mode	RW	-
	Enumerated values:		
	0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

XOSC: STARTUP Register

Offset: 0x0c

Description

Controls the startup delay

Table 262. STARTUP Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20	X4: Multiplies the startup_delay by 4. This is of little value to the user given that the delay can be programmed directly.	RW	0x0
19:14	Reserved.	-	-
13:0	DELAY: in multiples of 256*xtal_period. The reset value of 0xc4 corresponds to approx 50 000 cycles.	RW	0x00c4

XOSC: COUNT Register

Offset: 0x1c

Table 263. COUNT Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	A down counter running at the xosc frequency which counts to zero and stops. To start the counter write a non-zero value. Can be used for short software pauses when setting up time sensitive hardware.	RW	0x00

2.17. Ring Oscillator (ROSC)

2.17.1. Overview

The Ring Oscillator (ROSC) is an on-chip oscillator built from a ring of inverters. It requires no external components and is started automatically during RP2040 power up. It provides the clock to the cores during boot. The frequency of the

XOSC: 休眠模式寄存器

偏移量: 0x08

说明

晶体振荡器暂停控制

表 261. DORMANT
寄存器

位	描述	类型	复位值
31:0	此寄存器用于通过暂停XOSC以节省功耗 上电时该字段初始化为WAKE 无效写入同样会选择WAKE 警告：请选择休眠模式前必须停止PLL 警告：请选择休眠模式前须先设置中断请求	读写	-
	枚举值：		
	0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

XOSC: 启动寄存器

偏移: 0x0c

说明

控制启动延迟

表 262. STARTUP
寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20	X4: 将startup_delay乘以4。考虑到延迟可直接编程，用户对此值有限。 。	读写	0x0
19:14	保留。	-	-
13:0	DELAY: 以256*xtal_period的倍数计。复位值0xc4对应约50,000个周期。	读写	0x00c4

XOSC: COUNT寄存器

偏移量: 0x1c

表 263. COUNT
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	一个以xosc频率运行的递减计数器，计数至零后停止。 要启动计数器，请写入非零值。 可用于设置时间敏感硬件时的短时软件延迟。	读写	0x00

2.17. 环形振荡器 (ROSC)

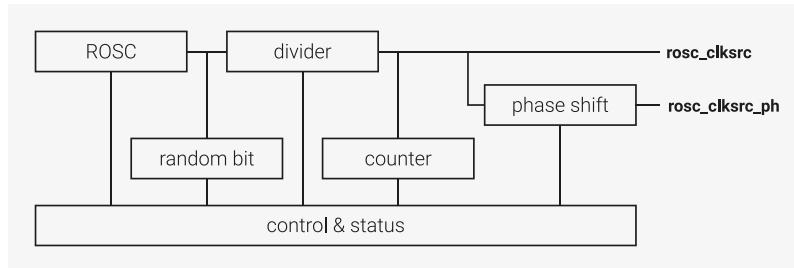
2.17.1. 概述

环形振荡器 (ROSC) 为由反相器环构成的片上振荡器。其无需外部元件，并在RP2040上电时自动启动，启动期间为核心提供时钟。

ROSC is programmable and it can directly provide a high speed clock to the cores, but the frequency varies with Process, Voltage and Temperature (PVT) so it cannot provide clocks for components which require an accurate frequency such as the RTC, USB and ADC. Methods for mitigating the frequency variation are discussed in [Section 2.15](#) but these are only relevant to very low power design. For most applications requiring accurate clock frequencies it is recommended to switch to the XOSC and PLLs. During boot the ROSC runs at a nominal 6.5MHz and is guaranteed to be in the range 1.8MHz to 12MHz.

Once the chip has booted the programmer can choose to continue running from the ROSC and increase its frequency or start the Crystal Oscillator (XOSC) and PLLs. The ROSC can be disabled after the system clocks have been switched to the XOSC. Each oscillator has advantages and the programmer can switch between them to achieve the best solution for the application.

Figure 34. ROSC overview.



2.17.2. ROSC/XOSC trade-offs

The advantages of the ROSC are its flexibility and its low power. Also, there is no requirement for internal or external components when using the ROSC to provide clocks. Its frequency is programmable so it can be used to provide a fast core clock without starting the PLLs and can be divided by clock generators ([Section 2.15](#)) to generate slower peripheral clocks. The ROSC starts immediately and responds immediately to the frequency controls. It will retain the frequency setting when entering and exiting the DORMANT state (see [Section 2.11.3](#)). However, the user must be aware that the frequency may have drifted when exiting the DORMANT state due to changes in the supply voltage and the chip temperature.

The disadvantage of the ROSC is its frequency variation with PVT (Process, Voltage & Temperature) which makes it unsuitable for generating precise clocks or for applications where software execution timing is important. However, the PVT frequency variation can be exploited to provide automatic frequency scaling to maximise performance. This is discussed in [Section 2.15](#).

The only advantage of the XOSC is its accurate frequency, but this is an overriding requirement in many applications.

The disadvantages of the XOSC are its requirement for external components (a crystal etc), its higher power consumption, slow startup time (>1ms) and fixed, low frequency. PLLs are required to produce higher frequency clocks. They consume more power and take significant time to start up and to change frequency. Exiting DORMANT mode is much slower than for ROSC because the XOSC must be restarted and the PLLs must be reconfigured.

2.17.3. Modifying the frequency

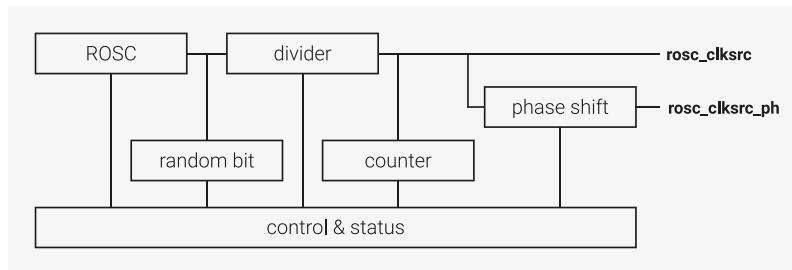
The ROSC is arranged as 8 stages, each with programmable drive. There are 2 methods of controlling the frequency. The frequency range controls the number of stages in the ROSC loop and the FREQA & FREQB registers control the drive strength of the stages.

The frequency range is changed by writing to the FREQ_RANGE register which controls the number of stages in the ROSC loop. The default LOW range has 8 (stages 0-7), MEDIUM has 6 (stages 2-7), HIGH has 4 (stages 4-7) and TOOHIGH has 2 (stages 6-7). It is recommended to change FREQ_RANGE one step at a time until the desired range is reached. The ROSC output will not glitch when increasing the frequency range, so the output clock can continue to be used. However, that is not true when going back down the frequency range. An alternate clock source must be selected for the modules clocked by ROSC, or they must be held in reset during the transition. The behaviour has not been fully characterised but the MEDIUM range will be approximately 1.33 times the LOW RANGE, the HIGH range will be 2 times

ROSC频率可编程，并可直接向核心提供高速时钟，但其频率随工艺、供电电压及温度（PVT）变化，因此无法为需精准频率的组件（如RTC、USB及ADC）提供时钟。第2.15节讨论了频率波动的缓解方案，仅适用于超低功耗设计。对大多数要求精确时钟频率的应用，建议切换至XOSC和PLL。ROSC启动时以标称6.5MHz运行，保证频率范围为1.8MHz至12MHz。

芯片启动后，程序员可选择继续使用ROSC运行并提高其频率，或启动晶体振荡器（XOSC）及PLL。系统时钟切换至XOSC后，ROSC可被禁用。各振荡器各具优势，程序员可根据应用需求切换使用，以实现最佳解决方案。

图34. ROSC
概述。



2.17.2. ROSC/XOSC 折衷

ROSC的优势在于其灵活性及低功耗。此外，使用ROSC提供时钟时，无需任何内部或外部组件。其频率可编程，可用于提供快速内核时钟而无需启动PLL，且可通过时钟发生器（见第2.15节）分频产生较慢的外设时钟。ROSC可立即启动，且即时响应频率控制。进入及退出休眠状态（见第2.11.3节）时，ROSC将保持频率设置。但用户须注意，由于电源电压和芯片温度变化，退出休眠状态时频率可能会漂移。

ROSC 的缺点在于其频率随着 PVT（工艺、电压和温度）的变化而变化，这使其不适合用于生成精确时钟或对软件执行时序要求严格的应用。然而，PVT 频率的变化可以被利用，实现自动频率调整，以最大化性能。相关内容详见第 2.15 节。

XOSC 的唯一优势在于其频率的准确性，这对许多应用而言是不可替代的要求。

XOSC 的缺点包括需要外部元件（如晶体）、较高的功耗、启动时间较长（>1ms）以及固定且较低的频率。必须使用 PLL 以产生更高频率的时钟信号。

PLL 功耗更高，启动及频率调整所需时间较长。退出休眠模式比 ROSC 慢得多，因需重新启动 XOSC 并重新配置 PLL。

2.17.3. 频率调整

ROSC 由 8 级组成，每级均具备可编程驱动能力。控制频率的方法有两种。

频率范围控制ROSC环路中的级数，FREQA和FREQB寄存器则控制各级的驱动强度。

通过写入FREQ_RANGE寄存器可以改变频率范围，该寄存器决定ROSC环路中的级数。默认情况下，LOW范围包含8级（第0至7级），MEDIUM为6级（第2至7级），HIGH为4级（第4至7级），TOOHIGH为2级（第6至7级）。建议逐步调整FREQ_RANGE，每次调整一步，直至达到所需的频率范围。增加频率范围时，ROSC输出不会出现毛刺，因此输出时钟可以持续使用。但在降低频率范围时，情况则不同。对于由ROSC时钟提供时钟信号的模块，必须选择备用时钟源，或在切换期间将其保持复位状态。该行为尚未完全描述，但中等范围大约是低范围的1.33倍，高范围将是低范围的2倍。

the LOW range and the TOOHIGH range will be 4 times the LOW range. The TOOHIGH range is aptly named. It should not be used because the internal logic of the ROSC will not run at that frequency.

The FREQA & FREQB registers control the drive strength of the stages in the ROSC loop. Increasing the drive strength reduces the delay through the stage and increases the oscillation frequency. Each stage has 3 drive strength control bits. Each bit turns on additional drive, therefore each stage has 4 drive strength settings equal to the number of bits set, with 0 being the default, 1 being double drive, 2 being triple drive and 3 being quadruple drive. Turning on extra drive will not have a linear effect on frequency, setting a second bit will have less impact than setting the first bit and so on. To ensure smooth transitions it is recommended to change one drive strength bit at a time. When FREQ_RANGE is used to shorten the ROSC loop, the bypassed stages still propagate the signal and therefore their drive strengths must be set to at least the same level as the lowest drive strength in the stages that are in the loop. This will not affect the oscillation frequency.

2.17.4. ROSC divider

The ROSC frequency is too fast to be used directly so is divided in an integer divider controlled by the DIV register. DIV can be changed while the ROSC is running, the output clock will change frequency without glitching. The default divisor is 16 which ensures the output clock is in the range 1.8 to 12MHz on chip startup.

The divider has 2 outputs, rosc_clksrc and rosc_clksrc_ph, the second being a phase shifted version of the first. This is primarily intended for use during product development and the outputs will be identical if the PHASE register is left in its default state.

2.17.5. Random Number Generator

If the system clocks are running from the XOSC and/or PLLs the ROSC can be used to generate random numbers. Simply enable the ROSC and read the RANDOMBIT register to get a 1-bit random number and read it n times to get an n-bit value. This does not meet the requirements of randomness for security systems because it can be compromised, but it may be useful in less critical applications. If the cores are running from the ROSC then the value will not be random because the timing of the register read will be correlated to the phase of the ROSC.

2.17.6. ROSC Counter

The COUNT register provides a method of managing short software delays. Writing a value to the COUNT register automatically triggers it to start counting down to zero at the ROSC frequency. The programmer then simply polls the register until it reaches zero. This is preferable to using NOPs in software loops because it is independent of the core clock frequency, the compiler and the execution time of the compiled code.

2.17.7. DORMANT mode

In DORMANT mode (see [Section 2.11.3](#)) all of the on-chip clocks can be paused to save power. This is particularly useful in battery-powered applications. The RP2040 is woken from DORMANT mode by an interrupt either from an external event such as an edge on a GPIO pin or from the on-chip RTC. This must be configured before entering DORMANT mode. If the RTC is being used to trigger wake-up then it must be clocked from an external source. To enter DORMANT mode the programmer must then switch all internal clocks to be driven from XOSC or ROSC and stop the PLLs. Then a specific 32-bit value must be written to the DORMANT register in the chosen oscillator (XOSC or ROSC) to stop it oscillating. When exiting DORMANT mode the chosen oscillator will restart. If XOSC is chosen then the frequency will be more precise but the restart time is longer due to the startup delay (>1ms on the RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#))). If ROSC is chosen then the frequency is less precise but the start-up time is very short (approximately 1μs).

低范围和过高范围将是低范围的4倍。“过高范围”称谓恰当。不应使用过高范围，因为ROSC的内部逻辑无法在该频率下正常运行。

FREQA与FREQB寄存器控制ROSC环路各阶段的驱动强度。增加驱动强度可缩短阶段延迟，提升振荡频率。每个阶段设有3个位的驱动强度控制位。每个位启用额外驱动，因此每个阶段共有4个驱动强度设定，取决于已置位数量：0为默认，1为双倍驱动，2为三倍驱动，3为四倍驱动。开启额外驱动对频率的影响非线性，第二个位的影响小于第一个，依此类推。

为确保过渡平稳，建议每次仅修改一个驱动强度位。当使用FREQ_RANGE缩短ROSC环路时，旁路的阶段仍会传递信号，因此其驱动强度必须设置为至少与环路中阶段的最低驱动强度相等。此操作不会影响振荡频率。

2.17.4. ROSC 分频器

ROSC频率过高，无法直接使用，因此通过由DIV寄存器控制的整数分频器进行分频。DIV可在ROSC运行时更改，输出时钟频率将无毛刺地变化。默认分频值为16，保证芯片启动时输出时钟频率处于1.8至12MHz范围内。

分频器设有两个输出，`rosc_clksrc`和`rosc_clksrc_ph`，后者为前者的相位移版本。此功能主要用于产品开发阶段，若PHASE寄存器保持默认状态，两个输出信号将完全相同。

2.17.5. 随机数生成器

如果系统时钟由XOSC和/或PLL驱动，则ROSC可用于生成随机数。

只需启用ROSC并读取RANDOMBIT寄存器，即可获得1位随机数，重复读取n次以获得n位随机数值。此方法无法满足安全系统对随机性的要求，因其可能被攻破，但在非关键应用中仍可使用。若核心运行于ROSC时钟，读取的值将不具随机性，因为寄存器读取的时序与ROSC的相位相关。

2.17.6. ROSC 计数器

COUNT寄存器提供了一种管理短时软件延迟的方法。向COUNT寄存器写入数值后，将自动以ROSC频率开始倒计时至零。程序员只需轮询该寄存器，直到其值达到零。此方法优于在软件循环中使用NOP指令，因为它不依赖核心时钟频率、编译器或编译后代码的执行时间。

2.17.7. 休眠模式

在休眠模式（详见第2.11.3节）下，所有片上时钟均可暂停以节省功耗。此功能在电池供电的应用中尤为重要。RP2040通过外部事件（如GPIO引脚的边沿变化）或片上RTC的中断唤醒休眠模式。此配置须在进入休眠模式前完成。如果实时钟（RTC）用于触发唤醒，则必须由外部时钟源提供时钟。要进入休眠模式，程序员必须将所有内部时钟切换为由XOSC或ROSC驱动，并停止PLL。随后，必须向所选振荡器（XOSC或ROSC）的DORMANT寄存器写入特定的32位数值以停止振荡。退出休眠模式时，所选振荡器将重新启动。若选择XOSC，则频率更为精准，但因启动延迟，重启时间较长（RP2040参考设计中超过1ms，详见《RP2040硬件设计》中的最简设计示例）。若选择ROSC，频率较不精准，但启动时间极短（约1μs）。

Pico Extras: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/hardware_rosc/rosc.c Lines 56 - 61

```

56 void rosc_set_dormant(void) {
57     // WARNING: This stops the rosc until woken up by an irq
58     rosc_write(&rosc_hw->dormant, ROSC_DORMANT_VALUE_DORMANT);
59     // Wait for it to become stable once woken up
60     while(!(rosc_hw->status & ROSC_STATUS_STABLE_BITS));
61 }
```

— WARNING

If no IRQ is configured before going into dormant mode the ROSC will never restart.

See [Section 2.11.5.2](#) for some examples of dormant mode.

2.17.8. List of Registers

The ROSC registers start at a base address of **0x40060000** (defined as **ROSC_BASE** in SDK).

Table 264. List of ROSC registers

Offset	Name	Info
0x00	CTRL	Ring Oscillator control
0x04	FREQA	Ring Oscillator frequency control A
0x08	FREQB	Ring Oscillator frequency control B
0x0c	DORMANT	Ring Oscillator pause control
0x10	DIV	Controls the output divider
0x14	PHASE	Controls the phase shifted output
0x18	STATUS	Ring Oscillator Status
0x1c	RANDOMBIT	Returns a 1 bit random value
0x20	COUNT	A down counter running at the ROSC frequency which counts to zero and stops.

ROSC: CTRL Register

Offset: 0x00

Description

Ring Oscillator control

Table 265. CTRL Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:12	ENABLE: On power-up this field is initialised to ENABLE The system clock must be switched to another source before setting this field to DISABLE otherwise the chip will lock up The 12-bit code is intended to give some protection against accidental writes. An invalid setting will enable the oscillator.	RW	-
	Enumerated values:		
	0xd1e → DISABLE		
	0xfb → ENABLE		

Pico附加资源: https://github.com/raspberrypi/pico-extras/blob/master/src/rp2_common/hardware_osc/osc.c 第56至61行

```

56 void rosc_set_dormant(void) {
57     // 警告：此操作将停止 rosc，直至通过中断唤醒
58     rosc_write(&rosc_hw->dormant, ROSC_DORMANT_VALUE_DORMANT);
59     // 等待唤醒后稳定
60     while(!(rosc_hw->status & ROSC_STATUS_STABLE_BITS));
61 }
```

● 警告

若在进入休眠模式前未配置IRQ，ROSC将无法重新启动。

有关休眠模式的部分示例，详见第2.11.5.2节。

2.17.8. 寄存器列表

ROSC寄存器起始地址为 **0x40060000** (SDK中定义为ROSC_BASE)。

表264.
ROSC寄存器列表

偏移量	名称	说明
0x00	CTRL	环振荡器控制
0x04	FREQA	环振荡器频率控制A
0x08	FREQB	环振荡器频率控制B
0x0c	休眠	环振荡器暂停控制
0x10	DIV	控制输出分频器
0x14	PHASE	控制相移输出
0x18	状态	环振荡器状态
0x1c	RANDOMBIT	返回1位随机值
0x20	计数	一个以ROSC频率运行的向下计数器，计数至零后停止。

ROSC: CTRL寄存器

偏移: 0x00

描述

环振荡器控制

表265. CTRL
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	ENABLE: 上电时此字段初始化为ENABLE 在将此字段设置为DISABLE之前，必须将系统时钟切换至其他时钟源，否则芯片将锁死 该12位代码旨在防止意外写入。 无效的设置将启用振荡器。	读写	-
	枚举值：		
	0xd1e → DISABLE		
	0xfb → ENABLE		

Bits	Description	Type	Reset
11:0	FREQ_RANGE: Controls the number of delay stages in the ROSC ring LOW uses stages 0 to 7 MEDIUM uses stages 2 to 7 HIGH uses stages 4 to 7 TOOHIGH uses stages 6 to 7 and should not be used because its frequency exceeds design specifications The clock output will not glitch when changing the range up one step at a time The clock output will glitch when changing the range down Note: the values here are gray coded which is why HIGH comes before TOOHIGH	RW	0xa0
	Enumerated values:		
	0xfa4 → LOW		
	0xfa5 → MEDIUM		
	0xfa7 → HIGH		
	0xfa6 → TOOHIGH		

ROSC: FREQA Register

Offset: 0x04

Description

The FREQA & FREQB registers control the frequency by controlling the drive strength of each stage
The drive strength has 4 levels determined by the number of bits set
Increasing the number of bits set increases the drive strength and increases the oscillation frequency
0 bits set is the default drive strength
1 bit set doubles the drive strength
2 bits set triples drive strength
3 bits set quadruples drive strength

Table 266. FREQA Register

Bits	Description	Type	Reset
31:16	PASSWD: Set to 0x9696 to apply the settings Any other value in this field will set all drive strengths to 0	RW	0x0000
	Enumerated values:		
	0x9696 → PASS		
15	Reserved.	-	-
14:12	DS3: Stage 3 drive strength	RW	0x0
11	Reserved.	-	-
10:8	DS2: Stage 2 drive strength	RW	0x0
7	Reserved.	-	-
6:4	DS1: Stage 1 drive strength	RW	0x0
3	Reserved.	-	-
2:0	DS0: Stage 0 drive strength	RW	0x0

ROSC: FREQB Register

Offset: 0x08

位	描述	类型	复位值
11:0	FREQ_RANGE: 控制ROSC环中的延迟阶段数 LOW使用第0至7阶段 MEDIUM使用第2至7阶段 HIGH使用第4至7阶段 TOOHIGH使用第6至7阶段，因其频率超过设计规格，故不应使用 时钟输出在逐级向上调整范围时不会产生毛刺 时钟输出在调整范围向下时会产生毛刺 注：此处数值采用格雷码，这也是HIGH排在TOOHIGH之前的原因	读写	0xaa0
	枚举值：		
	0xfa4 → LOW		
	0xfa5 → MEDIUM		
	0xfa7 → 高		
	0xfa6 → 过高		

ROSC: FREQA 寄存器

偏移量: 0x04

说明

FREQA 和 FREQB 寄存器通过控制各阶段的驱动强度来调节频率。驱动强度分为4级，由置位的位数决定。置位位数增加会增强驱动强度并提升振荡频率。

- 0 位置位为默认驱动强度
- 1 位置位使驱动强度加倍
- 2 位置位使驱动强度增加三倍
- 3 位置位使驱动强度增加四倍

表 266. FREQA 寄存器

位	描述	类型	复位值
31:16	PASSWD: 设置为 0x9696 以应用设置 该字段的任何其他数值均将使所有驱动强度归零	读写	0x0000
	枚举值：		
	0x9696 → 通过		
15	保留。	-	-
14:12	DS3: 第3阶段驱动强度	读写	0x0
11	保留。	-	-
10:8	DS2: 第2阶段驱动强度	读写	0x0
7	保留。	-	-
6:4	DS1: 第1阶段驱动强度	读写	0x0
3	保留。	-	-
2:0	DS0: 第0阶段驱动强度	读写	0x0

ROSC: FREQB 寄存器

偏移: 0x08

Description

For a detailed description see freqa register

Table 267. FREQB
Register

Bits	Description	Type	Reset
31:16	PASSWD: Set to 0x9696 to apply the settings Any other value in this field will set all drive strengths to 0	RW	0x0000
	Enumerated values: 0x9696 → PASS		
15	Reserved.	-	-
14:12	DS7: Stage 7 drive strength	RW	0x0
11	Reserved.	-	-
10:8	DS6: Stage 6 drive strength	RW	0x0
7	Reserved.	-	-
6:4	DS5: Stage 5 drive strength	RW	0x0
3	Reserved.	-	-
2:0	DS4: Stage 4 drive strength	RW	0x0

ROSC: DORMANT Register

Offset: 0x0c

Description

Ring Oscillator pause control

Table 268. DORMANT
Register

Bits	Description	Type	Reset
31:0	This is used to save power by pausing the ROSC On power-up this field is initialised to WAKE An invalid write will also select WAKE Warning: setup the irq before selecting dormant mode	RW	-
	Enumerated values: 0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

ROSC: DIV Register

Offset: 0x10

Description

Controls the output divider

Table 269. DIV
Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:0	set to 0xaa0 + div where div = 0 divides by 32 div = 1-31 divides by div any other value sets div=31 this register resets to div=16	RW	-
	Enumerated values:		

描述

详见 freqa 寄存器的详细描述

表 267。FREQB
寄存器

位	描述	类型	复位值
31:16	PASSWD : 设置为 0x9696 以应用设置 该字段的任何其他数值均将使所有驱动强度归零	读写	0x0000
	枚举值:		
	0x9696 → 通过		
15	保留。	-	-
14:12	DS7 : 第 7 级驱动强度	读写	0x0
11	保留。	-	-
10:8	DS6 : 第 6 级驱动强度	读写	0x0
7	保留。	-	-
6:4	DS5 : 第 5 级驱动强度	读写	0x0
3	保留。	-	-
2:0	DS4 : 第 4 级驱动强度	读写	0x0

ROSC：休眠模式寄存器

偏移: 0x0c

说明

环振荡器暂停控制

表 268。休眠模式
寄存器

位	描述	类型	复位值
31:0	该寄存器用于通过暂停 ROSC 来节省功耗 上电时该字段初始化为 WAKE 无效写入同样会选择 WAKE 警告: 请在选择休眠模式之前设置 irq	读写	-
	枚举值:		
	0x636f6d61 → DORMANT		
	0x77616b65 → WAKE		

ROSC：DIV 寄存器

偏移: 0x10

说明

控制输出分频器

表 269。DIV
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	设置为 $0xaaa + div$, 其中 $div = 0$ 表示除以 32 $div = 1-31$ 表示除以 div 其他值均设定 $div = 31$ 该寄存器复位时, $div = 16$	读写	-
	枚举值:		

Bits	Description	Type	Reset
	0xaa0 → PASS		

ROSC: PHASE Register

Offset: 0x14

Description

Controls the phase shifted output

Table 270. PHASE Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:4	PASSWD : set to 0xaa any other value enables the output with shift=0	RW	0x00
3	ENABLE : enable the phase-shifted output this can be changed on-the-fly	RW	0x1
2	FLIP : invert the phase-shifted output this is ignored when div=1	RW	0x0
1:0	SHIFT : phase shift the phase-shifted output by SHIFT input clocks this can be changed on-the-fly must be set to 0 before setting div=1	RW	0x0

ROSC: STATUS Register

Offset: 0x18

Description

Ring Oscillator Status

Table 271. STATUS Register

Bits	Description	Type	Reset
31	STABLE : Oscillator is running and stable	RO	0x0
30:25	Reserved.	-	-
24	BADWRITE : An invalid value has been written to CTRL_ENABLE or CTRL_FREQ_RANGE or FREQA or FREQB or DIV or PHASE or DORMANT	WC	0x0
23:17	Reserved.	-	-
16	DIV_RUNNING : post-divider is running this resets to 0 but transitions to 1 during chip startup	RO	-
15:13	Reserved.	-	-
12	ENABLED : Oscillator is enabled but not necessarily running and stable this resets to 0 but transitions to 1 during chip startup	RO	-
11:0	Reserved.	-	-

ROSC: RANDOMBIT Register

Offset: 0x1c

Table 272.
RANDOMBIT Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-

位	描述	类型	复位值
	0xaa0 → 通过		

ROSC: PHASE 寄存器

偏移: 0x14

说明

控制相移输出

表270. PHASE 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:4	PASSWD : 设置为0xaa 其他任意值启用输出且shift=0	读写	0x00
3	ENABLE : 启用相位偏移输出 该项支持动态修改	读写	0x1
2	FLIP : 反转相位偏移输出 当div=1时，将忽略此项	读写	0x0
1:0	SHIFT : 对相位偏移输出进行SHIFT输入时钟的相位移 该项支持动态修改 设置div=1前，须先将其设为0	读写	0x0

ROSC: STATUS 寄存器

偏移: 0x18

描述

环振荡器状态

表271. STATUS 寄存器

位	描述	类型	复位值
31	STABLE : 振荡器正在运行且稳定	只读	0x0
30:25	保留。	-	-
24	BADWRITE : 向CTRL_ENABLE、CTRL_FREQ_RANGE、FREQA、 FREQB、DIV、PHASE或DORMANT寄存器写入了无效值	WC	0x0
23:17	保留。	-	-
16	DIV_RUNNING : 后置分频器正在运行 该值复位为0，但芯片启动期间会变为1	只读	-
15:13	保留。	-	-
12	ENABLED : 振荡器已启用，但不一定正在运行或处于稳定状态 该值复位为0，但芯片启动期间会变为1	只读	-
11:0	保留。	-	-

ROSC: RANDOMBIT 寄存器

偏移量: 0x1c

表272。
RANDOMBIT 寄存器

位	描述	类型	复位值
31:1	保留。	-	-

Bits	Description	Type	Reset
0	This just reads the state of the oscillator output so randomness is compromised if the ring oscillator is stopped or run at a harmonic of the bus frequency	RO	0x1

ROSC: COUNT Register

Offset: 0x20

Table 273. COUNT Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	A down counter running at the ROSC frequency which counts to zero and stops. To start the counter write a non-zero value. Can be used for short software pauses when setting up time sensitive hardware.	RW	0x00

2.18. PLL

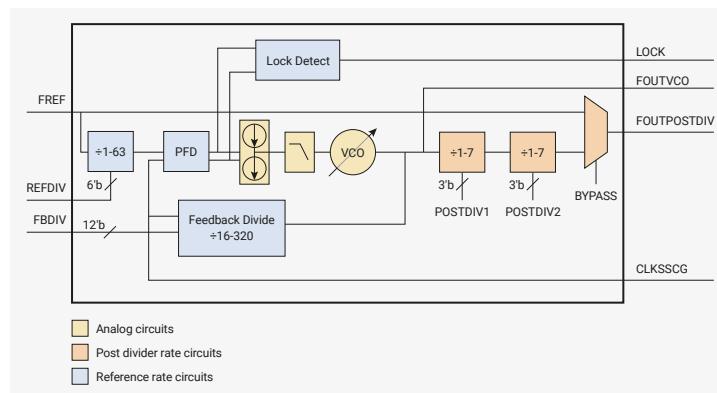
2.18.1. Overview

The PLL is designed to take a reference clock, and multiply it using a VCO (Voltage Controlled Oscillator) with a feedback loop. The VCO must run at high frequencies (between 750 and 1600MHz), so there are two dividers, known as post dividers that can divide the VCO frequency before it is distributed to the clock generators on the chip.

There are two PLLs in RP2040. They are:

- `pll_sys` - Used to generate up to a 133MHz system clock
- `pll_usb` - Used to generate a 48MHz USB reference clock

Figure 35. On both PLLs, the FREF (reference) input is connected to the crystal oscillator's X1 input. The PLL contains a VCO, which is locked to a constant ratio of the reference clock via the feedback loop (phase-frequency detector and loop filter). This can synthesise very high frequencies, which may be divided down by the post-dividers.



2.18.2. Calculating PLL parameters

To configure the PLL, you must know the frequency of the reference clock, which on RP2040 is routed directly from the crystal oscillator. This will often be a 12MHz crystal, for compatibility with RP2040's USB bootrom. The PLL's final output frequency `FOUTPOSTDIV` can then be calculated as $(\text{FREF} / \text{REFDIV}) \times \text{FBDIV} / (\text{POSTDIV1} \times \text{POSTDIV2})$. With a desired output frequency in mind, you must select PLL parameters according to the following constraints of the PLL design:

- Minimum reference frequency (`FREF / REFDIV`) is 5MHz

位	描述	类型	复位值
0	该寄存器仅读取振荡器输出状态，因此当环形振荡器停止或以总线频率的谐波运行时，随机性将受损。	只读	0x1

ROSC: COUNT 寄存器

偏移量: 0x20

表 273。COUNT
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	一个以ROSC频率运行的向下计数器，计数至零后停止。 要启动计数器，请写入非零值。 可用于设置时间敏感硬件时的短时软件延迟。	读写	0x00

2.18. PLL

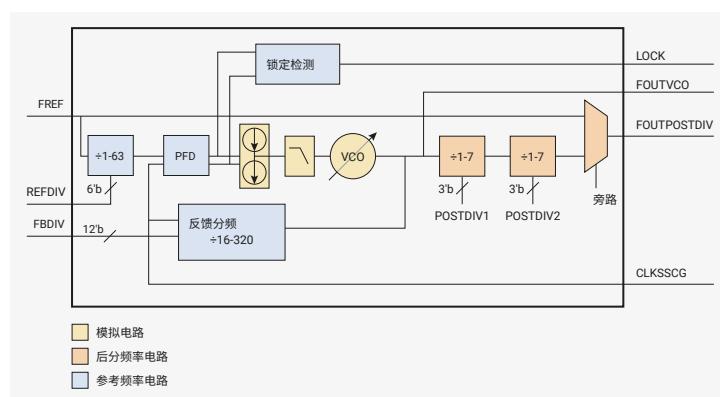
2.18.1. 概述

PLL 设计用于接收参考时钟，并通过带有反馈环路的压控振荡器（VCO）进行倍频。VCO 必须在高频率范围运行（750 至 1600 MHz），因此设有两个后置分频器，可在将 VCO 频率分配给芯片上的时钟发生器之前对其进行分频。

RP2040 内置两个 PLL，如下所示：

- **pll_sys** - 用于生成最高 133 MHz 的系统时钟
- **pll_usb** - 用于生成 48 MHz 的 USB 参考时钟

图35。两个PLL 中，REF（参考）输入均连接至晶体振荡器的X输入端。PL 包含一个VC O，通过反馈环路（相位频率检测器和环路滤波器）将其锁定至参考时钟的恒定倍数。该设计能够合成极高的频率，随后可由后分频器进行降频。



2.18.2. PLL 参数计算

配置PLL时，必须了解参考时钟频率，RP2040中该频率直接取自晶体振荡器。该参考时钟通常为12MHz晶体，以兼容RP 2040的USB引导只读存储器。PLL的最终输出频率FOUTPOSTDIV可按公式计算： $(\text{FREF} / \text{REFDIV}) \times \text{FB DIV} / (\text{POSTDIV1} \times \text{POSTDIV2})$ 。基于所需的输出频率，必须依照以下PLL设计约束选择PLL参数：

- 最小参考频率 (**FREF / REFDIV**) 为5MHz

- Oscillator frequency (**FOUTVCO**) must be in the range 750MHz → 1600MHz
- Feedback divider (**FBDIV**) must be in the range 16 → 320
- The post dividers **POSTDIV1** and **POSTDIV2** must be in the range 1 → 7
- Maximum input frequency (**FREF / REFDIV**) is VCO frequency divided by 16, due to minimum feedback divisor

Additionally, the maximum frequencies of the chip's clock generators (attached to **FOUTPOSTDIV**) must be respected. For the system PLL this is 133MHz, and for the USB PLL, 48MHz.

NOTE

The crystal oscillator on RP2040 is designed for crystals between 5 and 15MHz, so typically **REFDIV** should be 1. If the application circuit drives a faster reference directly into the XI input, and a low VCO frequency is desired, the reference divisor can be increased to keep the PLL input within a suitable range.

TIP

When two different values are required for **POSTDIV1** and **POSTDIV2**, it's preferable to assign the higher value to **POSTDIV1**, for lower power consumption.

In the RP2040 reference design (see the Minimal Design Example in [Hardware design with RP2040](#)), which attaches a 12MHz crystal to the crystal oscillator, this implies that the minimum achievable and legal VCO frequency is $12\text{MHz} \times 63 = 756\text{MHz}$, and the maximum VCO is $12\text{MHz} \times 133 = 1596\text{MHz}$, so **FBDIV** must remain in the range 63 → 133. For example, setting **FBDIV** to 100 would synthesise a 1200MHz VCO frequency. A **POSTDIV1** value of 6 and a **POSTDIV2** value of 2 would divide this by 12 in total, producing a clean 100MHz at the PLL's final output.

2.18.2.1. Jitter vs Power Consumption

There are often several sets of PLL configuration parameters which achieve, or are very close to, the desired output frequency. It is up to the programmer to decide whether to prioritise low PLL power consumption, or lower *jitter*, which is cycle-to-cycle variation in the PLL's output clock period. This is not a concern as far as system stability is concerned, because RP2040's digital logic is designed with margin for the worst-case possible jitter on the system clock, but a highly accurate clock is often needed for audio and video applications, or where data is being transmitted and received in accordance with a specification. For example, the USB specification defines a maximum amount of allowable jitter.

Jitter is minimised by running the VCO at the highest possible frequency, so that higher post-divide values can be used. For example, $1500\text{MHz VCO} / 6 / 2 = 125\text{MHz}$. To reduce power consumption, the VCO frequency should be as low as possible. For example: $750\text{MHz VCO} / 6 / 1 = 125\text{MHz}$.

Another consideration here is that slightly adjusting the output frequency may allow a much lower VCO frequency to be achieved, by bringing the output to a closer rational multiple of the input. Indeed the exact desired frequency may not be exactly achievable with any allowable VCO frequency, or combination of divisors.

SDK provides a Python script that searches for the best VCO and post divider options for a desired output frequency:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/scripts/vcocalc.py

```

1 #!/usr/bin/env python3
2
3 import argparse
4 import sys
5
6 # Fixed hardware parameters
7 fbdv_range = range(16, 320 + 1)
8 postdiv_range = range(1, 7 + 1)
9 ref_min = 5
10 refdiv_min = 1
11 refdiv_max = 63

```

- 振荡器频率 (**FOUTVCO**) 须处于 $750\text{MHz} \rightarrow 1600\text{MHz}$ 范围内
- 反馈分频器 (**FBDIV**) 须处于 $16 \rightarrow 320$ 范围内
- 后置分频器 **POSTDIV1**和 **POSTDIV2**须处于 $1 \rightarrow 7$ 范围内
- 最大输入频率 (**FREF / REFDIV**) 为VCO频率除以16, 此因最小反馈分频器限制

此外, 芯片时钟生成器 (连接至**FOUTPOSTDIV**) 的最大频率亦须被遵守。系统PLL为 133MHz , USB PLL为 48MHz 。

注意

RP2040上的晶体振荡器设计适用于5至 15MHz 范围的晶体, 因此通常 **REFDIV**应设为1。如果应用电路直接驱动更高速率参考信号进入XI输入, 且希望获得较低的VCO频率, 则可以增大参考分频, 以保持PLL输入处于适宜范围。

提示

当 **POSTDIV1**和 **POSTDIV2**需要赋予不同数值时, 建议将较大数值赋予 **POSTDIV1**, 以实现更低的功耗。

在RP2040参考设计中 (参见《RP2040硬件设计》中的极简设计示例), 其将一枚 12MHz 晶体连接至晶体振荡器, 这意味着最低可实现且合法的VCO频率为 $12\text{MHz} \times 63 = 756\text{MHz}$, 且最大VCO频率为 $12\text{MHz} \times 133 = 1596\text{MHz}$, 因此 **FBDIV**必须保持在 $63 \rightarrow 133$ 范围内。例如, 将 **FBDIV**设定为100将合成出 1200MHz 的VCO频率。若 **POSTDIV1**设为6, **POSTDIV2**设为2, 则总体分频为12, 从而在PLL最终输出端产生稳定的 100MHz 信号。

2.18.2.1. 抖动与功耗

通常存在若干组PLL配置参数能够实现或非常接近所需的输出频率。程序员需决定是优先降低PLL功耗, 还是降低抖动—即PLL输出时钟周期之间的变化。这对系统稳定性不构成影响, 因为RP2040的数字逻辑设计中预留了系统时钟最坏情况抖动的裕量, 但在音频、视频应用或依据规范进行数据传输时, 通常需要极高精度的时钟。例如, USB规范规定了允许的最大抖动值。

抖动通过使VCO以尽可能高的频率运行, 从而采用更高的后分频值得以最小化。

例如, 1500MHz 的VCO经过6和2的连续分频后得到 125MHz 。为降低功耗, VCO频率应尽可能降低。例如: $750\text{MHz} \text{ VCO} / 6 / 1 = 125\text{MHz}$ 。

另一个需要考虑的问题是, 通过将输出频率调整至接近输入频率的有理数倍, 可能实现更低的VCO频率。实际上, 任何允许的VCO频率或除数组合可能无法精确达到所需的确切频率。

SDK提供了一个Python脚本, 用以搜索满足目标输出频率的最佳VCO和后置分频选项:

SDK链接: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/scripts/vcocalc.py

```

1 #!/usr/bin/env python3
2
3 import argparse
4 import sys
5
6 #固定硬件参数
7 fbdiv_range = range(16, 320 + 1)
8 postdiv_range = range(1, 7 + 1)
9 ref_min = 5
10 refdiv_min = 1
11 refdiv_max = 63

```

```

12
13 def validRefdiv(string):
14     if ((int(string) < refdiv_min) or (int(string) > refdiv_max)):
15         raise ValueError("REFDIV must be in the range {} to {}".format(refdiv_min,
16             refdiv_max))
17     return int(string)
18
19 parser = argparse.ArgumentParser(description="PLL parameter calculator")
20 parser.add_argument("--input", "-i", default=12, help="Input (reference) frequency. Default
21 12 MHz", type=float)
22 parser.add_argument("--ref-min", default=5, help="Override minimum reference frequency.
23 Default 5 MHz", type=float)
24 parser.add_argument("--vco-max", default=1600, help="Override maximum VCO frequency. Default
25 1600 MHz", type=float)
26 parser.add_argument("--vco-min", default=750, help="Override minimum VCO frequency. Default
27 750 MHz", type=float)
28 parser.add_argument("--cmake", action="store_true", help="Print out a CMake snippet to apply
29 the selected PLL parameters to your program")
30 parser.add_argument("--cmake-only", action="store_true", help="Same as --cmake, but do not
31 print anything other than the CMake output")
32 parser.add_argument("--cmake-executable-name", default=<program>, help="Set the executable
33 name to use in the generated CMake output")
34 parser.add_argument("--lock-refdiv", help="Lock REFDIV to specified number in the range {} to
35 {}".format(refdiv_min, refdiv_max), type=validRefdiv)
36 parser.add_argument("--low-vco", "-l", action="store_true", help="Use a lower VCO frequency
37 when possible. This reduces power consumption, at the cost of increased jitter")
38 parser.add_argument("output", help="Output frequency in MHz.", type=float)
39 args = parser.parse_args()
40
41 refdiv_range = range(refdiv_min, max(refdiv_min, min(refdiv_max, int(args.input / args
42 .ref_min))) + 1)
43 if args.lock_refdiv:
44     print("Locking REFDIV to", args.lock_refdiv)
45     refdiv_range = [args.lock_refdiv]
46
47 best = (0, 0, 0, 0, 0, 0)
48 best_margin = args.output
49
50 for refdiv in refdiv_range:
51     for fbdऱ in fbdऱ_range:
52         vco = args.input / refdiv * fbdऱ
53         if vco < args.vco_min or vco > args.vco_max:
54             continue
55         # pd1 is inner loop so that we prefer higher ratios of pd1:pd2
56         for pd2 in postdiv_range:
57             for pd1 in postdiv_range:
58                 out = vco / pd1 / pd2
59                 margin = abs(out - args.output)
60                 vco_is_better = vco < best[5] if args.low_vco else vco > best[5]
61                 if ((vco * 1000) % (pd1 * pd2)):
62                     continue
63                 if margin < best_margin or (abs(margin - best_margin) < 1e-9 and
64                 vco_is_better):
65                     best = (out, fbdऱ, pd1, pd2, refdiv, vco)
66                     best_margin = margin
67
68 best_out, best_fbdऱ, best_pd1, best_pd2, best_refdiv, best_vco = best
69
70 if best[0] > 0:
71     cmake_output = \
72     """target_compile_definitions({args.cmake_executable_name} PRIVATE
73     PLL_SYS_REFDIV={best_refdiv}
74     PLL_SYS_VCO_FREQ_HZ={int((args.input * 1_000_000) / best_refdiv * best_fbdऱ)}
75     PLL_SYS_POSTDIV1={best_pd1}"""

```

```

12
13 def validRefdiv(string):
14     if ((int(string) < refdiv_min) or (int(string) > refdiv_max)):
15         raise ValueError("REFDIV 必须位于 {} 至 {} 范围内".format(refdiv_min, refdiv_max))
16     return int(string)
17
18 parser = argparse.ArgumentParser(description="PLL 参数计算器")
19 parser.add_argument("--input", "-i", default=12, help="输入 (参考) 频率, 默认 12 MHz", type=float)
20 parser.add_argument("--ref-min", default=5, help="覆盖最小参考频率, 默认 5 MHz", type=float)
21 parser.add_argument("--vco-max", default=1600, help="覆盖最大 VCO 频率, 默认 1600 MHz", type=float)
22 parser.add_argument("--vco-min", default=750, help="覆盖最小 VCO 频率, 默认 750 MHz", type=float)
23 parser.add_argument("--cmake", action="store_true", help="打印CMake代码片段以将所选PLL参数应用于您的程序")
24 parser.add_argument("--cmake-only", action="store_true", help="功能同--cmake, 但仅输出CMake内容, 不打印其他信息")
25 parser.add_argument("--cmake-executable-name", default=<program>, help="设置生成CMake输出时使用的可执行文件名称")
26 parser.add_argument("--lock-refdiv", help="将REFDIV锁定为范围{} 至{}内的指定数值".format(refdiv_min, refdiv_max), type=validRefdiv)
27 parser.add_argument("--low-vco", "-l", action="store_true", help="在可能时使用较低的VCO频率, 以降低功耗, 但会增加抖动")
28 parser.add_argument("output", help="输出频率 (MHz)", type=float)
29 args = parser.parse_args()
30
31 refdiv_range = range(refdiv_min, max(refdiv_min, min(refdiv_max, int(args.input / args.ref_min))) + 1)
32 if args.lock_refdiv:
33     print("锁定REFDIV为", args.lock_refdiv)
34     refdiv_range = [args.lock_refdiv]
35
36 best = (0, 0, 0, 0, 0, 0)
37 best_margin = args.output
38
39 for refdiv in refdiv_range:
40     for fbdv in fbdv_range:
41         vco = args.input / refdiv * fbdv
42         if vco < args.vco_min or vco > args.vco_max:
43             continue
44         # pd1 是内层循环, 因此优先更高的 pd1:pd2 比率
45         for pd2 in postdiv_range:
46             for pd1 in postdiv_range:
47                 out = vco / pd1 / pd2
48                 margin = abs(out - args.output)
49                 vco_is_better = vco < best[5] if args.low_vco else vco > best[5]
50                 if ((vco * 1000) % (pd1 * pd2)):
51                     continue
52                 if margin < best_margin or (abs(margin - best_margin) < 1e-9 and
53                     vco_is_better):
54                     best = (out, fbdv, pd1, pd2, refdiv, vco)
55                     best_margin = margin
56
57 best_out, best_fbdv, best_pd1, best_pd2, best_refdiv, best_vco = best
58
59 if best[0] > 0:
60     cmake_output = \
61     """target_compile_definitions({args.cmake_executable_name} PRIVATE
62     PLL_SYS_REFDIV={best_refdiv}
63     PLL_SYS_VCO_FREQ_HZ={int((args.input * 1_000_000) / best_refdiv * best_fbdv)}
64     PLL_SYS_POSTDIV1={best_pd1}"""

```

```

64     PLL_SYS_POSTDIV2={best_pd2}
65     SYS_CLK_HZ=(int((args.input * 1_000_000) / (best_refdiv * best_pd1 * best_pd2) *
66     best_fbddiv)}
67 """
68     if not args.cmake_only:
69         print("Requested: {} MHz".format(args.output))
70         print("Achieved: {} MHz".format(best_out))
71         print("REFDIV:    {}".format(best_refdiv))
72         print("FBDIV:    {} (VCO = {} MHz)".format(best_fbddiv, args.input / best_refdiv *
73             best_fbddiv))
74         print("PD1:      {}".format(best_pd1))
75         print("PD2:      {}".format(best_pd2))
76         if best_refdiv != 1:
77             print(
78                 "\nThis requires a non-default REFDIV value.\n"
79                 "Add the following to your CMakeLists.txt to apply the REFDIV:\n"
80             )
81     elif args.cmake or args.cmake_only:
82         print("")
83     if args.cmake or args.cmake_only or best_refdiv != 1:
84         print(cmake_output)
85 else:
86     sys.exit("No solution found")

```

Given an input and output frequency, this script will find the best possible set of PLL parameters to get as close as possible. Where multiple equally good combinations are found, it returns the parameters which yield the highest VCO frequency, for best output stability. The `-l` or `--low-vco` flag will instead prefer lower frequencies, for reduced power consumption.

Here a 48MHz output is requested:

```

$ ./vcocalc.py 48
Requested: 48.0 MHz
Achieved: 48.0 MHz
FBDIV: 120 (VCO = 1440 MHz)
PD1: 6
PD2: 5

```

Asking for a 48MHz output with a lower VCO frequency, if possible:

```

$ ./vcocalc.py -l 48
Requested: 48.0 MHz
Achieved: 48.0 MHz
FBDIV: 64 (VCO = 768 MHz)
PD1: 4
PD2: 4

```

For a 125MHz system clock with a 12MHz input, the minimum VCO frequency is quite high.

```

$ ./vcocalc.py -l 125
Requested: 125.0 MHz
Achieved: 125.0 MHz
FBDIV: 125 (VCO = 1500 MHz)
PD1: 6

```

```

64     PLL_SYS_POSTDIV2={best_pd2}
65     SYS_CLK_HZ=(int((args.input * 1_000_000) / (best_refdiv * best_pd1 * best_pd2) *
66     best_fbddiv)}
67 """
68     if not args.cmake_only:
69         print("请求频率: {} MHz".format(args.output))
70         print("实现频率: {} MHz".format(best_out))
71         print("REFDIV:    {}".format(best_refdiv))
72         print("FBDIV:    {} (VCO = {} MHz)".format(best_fbddiv, args.input / best_refdiv *
73             best_fbddiv))
74         print("PD1:      {}".format(best_pd1))
75         print("PD2:      {}".format(best_pd2))
76         if best_refdiv != 1:
77             print(
78                 "\n这需要非默认的 REFDIV 值。\\n"
79                 "请将以下内容添加至您的 CMakeLists.txt 以应用该 REFDIV:\\n"
80             )
81     elif args.cmake 或 args.cmake_only:
82         print("")
83         if args.cmake 或 args.cmake_only 或 best_refdiv != 1:
84             print(cmake_output)
85 else:
86     sys.exit("未找到解决方案")

```

给定输入和输出频率，本脚本将寻找最佳PLL参数组合，以尽可能接近目标频率。当存在多个同样优良的组合时，返回能产生最高VCO频率的参数，以保证最佳输出稳定性。使用 **-l** 或 **--low-vco** 选项将优先选择较低频率，以降低功耗。

此处请求一个48 MHz输出：

```

$ ./vcocalc.py 48
请求频率: 48.0 MHz
实现频率: 48.0 MHz
FBDIV: 120 (VCO = 1440 MHz)
PD1: 6
PD2: 5

```

若可能，希望以较低VCO频率输出48 MHz：

```

$ ./vcocalc.py -l 48
请求频率: 48.0 MHz
实现频率: 48.0 MHz
FBDIV: 64 (VCO = 768 MHz)
PD1: 4
PD2: 4

```

对于12MHz输入信号的125MHz系统时钟，最低VCO频率相当高。

```

$ ./vcocalc.py -l 125
请求频率: 125.0 MHz
实际频率: 125.0 MHz
FBDIV: 125 (VCO = 1500 MHz)
PD1: 6

```

PD2 : 2

We can restrict the search to lower VCO frequencies, so that the script will consider looser frequency matches. Note that, whilst a 750MHz VCO would be ideal here, we can't achieve exactly 750MHz by multiplying the 12MHz input by an integer, which is why the previous invocation returned such a high VCO frequency.

```
$ ./vcocalc.py -l 125 --vco-max 800
Requested: 125.0 MHz
Achieved: 126.0 MHz
FBDIV: 63 (VCO = 756 MHz)
PD1: 6
PD2: 1
```

A 126MHz system clock may be a tolerable deviation from the desired 125MHz, and generating this clock consumes less power at the PLL.

2.18.3. Configuration

The SDK uses the following PLL settings:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware/clocks.h Lines 143 - 164

```
143 // There are two PLLs in RP-series microcontrollers:
144 // 1. The 'SYS PLL' generates the system clock, the frequency is defined by `SYS_CLK_KHZ`.
145 // 2. The 'USB PLL' generates the USB clock, the frequency is defined by `USB_CLK_KHZ`.
146 //
147 // The two PLLs use the crystal oscillator output directly as their reference frequency input;
// the PLLs reference
148 // frequency cannot be reduced by the dividers present in the clocks block. The crystal
// frequency is defined by `XOSC_HZ` (or
149 // `XOSC_KHZ` or `XOSC_MHZ`).
150 //
151 // The system's default definitions are correct for the above frequencies with a 12MHz
152 // crystal frequency. If different frequencies are required, these must be defined in
153 // the board configuration file together with the revised PLL settings
154 // Use `vcocalc.py` to check and calculate new PLL settings if you change any of these
// frequencies.
155 //
156 // Default PLL configuration RP2040:
157 //           REF      FBDIV   VCO          POSTDIV
158 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 6 / 2 = 125MHz
159 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 = 48MHz
160 //
161 // Default PLL configuration RP2350:
162 //           REF      FBDIV   VCO          POSTDIV
163 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 5 / 2 = 150MHz
164 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 = 48MHz
```

The `pll_init` function in the SDK, which we will examine below, asserts that all of these conditions are true before attempting to configure the PLL.

The SDK defines the PLL control registers as a struct. It then maps them into memory for each instance of the PLL.

PD2 : 2

我们可以将搜索范围限制在较低的VCO频率内，使脚本考虑更宽松的频率匹配。请注意，尽管750MHz的VCO频率在此处较为理想，但由于无法通过将12MHz输入信号乘以整数得到精确的750MHz，因此之前的调用返回了较高的VCO频率。

```
$ ./vcocalc.py -l 125 --vco-max 800
请求频率: 125.0 MHz
实际频率: 126.0 MHz
FB DIV: 63 (VCO = 756 MHz)
PD1: 6
PD2: 1
```

126MHz的系统时钟可能是对目标125MHz的可接受偏差，且在PLL中生成此时钟时功耗较低。

2.18.3. 配置

SDK使用以下PLL设置：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_clocks/include/hardware/clocks.h 第143至164行

```
143 // RP 系列微控制器中有两个PLL：
144 // 1. 'SYS PLL' 用于生成系统时钟，频率由 `SYS_CLK_KHZ` 定义。
145 // 2. 'USB PLL' 用于生成USB时钟，频率由 `USB_CLK_KHZ` 定义。
146 //
147 // 两个PLL 均直接使用晶体振荡器的输出作为其参考频率输入；
// PLL 的参考
148 // 频率不能被时钟模块中的分频器降低。晶体振荡器频率由 `XOSC_HZ` 定义（或 `XOSC_KHZ` 或 `XOSC_MHZ`）。

150 //
151 // 系统默认定义适用于上述频率，晶体频率为12MHz
152 // 晶体频率。若需不同频率，必须在
153 // 板级配置文件中定义，并同时调整PLL 设置。
154 // 如果更改任一频率，请使用 `vcocalc.py` 检查并计算新的PLL 设置。

155 //
156 // RP2040默认PLL配置：
157 //           REF      FB DIV VCO          POSTDIV
158 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 6 / 2 = 125MHz
159 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 =        48MHz
160 //
161 // RP2350默认PLL配置：
162 //           REF      FB DIV VCO          POSTDIV
163 // PLL SYS: 12 / 1 = 12MHz * 125 = 1500MHz / 5 / 2 = 150MHz
164 // PLL USB: 12 / 1 = 12MHz * 100 = 1200MHz / 5 / 5 =        48MHz
```

SDK 中的 `pll_init` 函数（我们将在下文详细审查）断言所有这些条件均成立后，方尝试配置 PLL。

SDK 将 PLL 控制寄存器定义为结构体。随后，将其映射至每个 PLL 实例的内存中。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h Lines 27 - 53

```

27 typedef struct {
28     _REG_(PLL_CS_OFFSET) // PLL_CS
29     // Control and Status
30     // 0x80000000 [31]    LOCK      (0) PLL is locked
31     // 0x00000100 [8]     BYPASS    (0) Passes the reference clock to the output instead of
32     // the...
33     // 0x0000003f [5:0]   REFDIV    (0x01) Divides the PLL input reference clock
34     io_rw_32 cs;
35
36     _REG_(PLL_PWR_OFFSET) // PLL_PWR
37     // Controls the PLL power modes
38     // 0x00000020 [5]     VCOPD    (1) PLL VCO powerdown +
39     // 0x00000008 [3]     POSTDIVPD (1) PLL post divider powerdown +
40     // 0x00000004 [2]     DSMRD    (1) PLL DSM powerdown +
41     // 0x00000001 [0]     PD        (1) PLL powerdown +
42     io_rw_32 pwr;
43
44     _REG_(PLL_FBDIV_INT_OFFSET) // PLL_FBDIV_INT
45     // Feedback divisor
46     // 0x00000fff [11:0]  FB DIV INT (0x000) see ctrl reg description for constraints
47     io_rw_32 fbdv_int;
48
49     _REG_(PLL_PRIM_OFFSET) // PLL_PRIM
50     // Controls the PLL post dividers for the primary output
51     // 0x00070000 [18:16] POSTDIV1 (0x7) divide by 1-7
52     // 0x00007000 [14:12] POSTDIV2 (0x7) divide by 1-7
53     io_rw_32 prim;
54 } pll_hw_t;
```

The SDK defines `pll_init` which is used to configure, or reconfigure a PLL. It starts by clearing any previous power state in the PLL, then calculates the appropriate feedback divider value. There are assertions to check these values satisfy the constraints above.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pll/pll.c Lines 13 - 21

```

13 void pll_init(PLL pll, uint refdiv, uint vco_freq, uint post_div1, uint post_div2) {
14     uint32_t ref_freq = XOSC_HZ / refdiv;
15
16     // Check vco freq is in an acceptable range
17     assert(vco_freq >= PICO_PLL_VCO_MIN_FREQ_HZ && vco_freq <= PICO_PLL_VCO_MAX_FREQ_HZ);
18
19     // What are we multiplying the reference clock by to get the vco freq
20     // (The regs are called div, because you divide the vco output and compare it to the
21     // refclk)
21     uint32_t fbdiv = vco_freq / ref_freq;
```

The programming sequence for the PLL is as follows:

- Program the reference clock divider (is a divide by 1 in the RP2040 case)
- Program the feedback divider
- Turn on the main power and VCO
- Wait for the VCO to lock (i.e. keep its output frequency stable)
- Set up post dividers and turn them on

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h 第27行至53行

```

27 typedef struct {
28     _REG_(PLL_CS_OFFSET) // PLL_CS
29     // 控制与状态
30     // 0x80000000 [31]      LOCK        (0) PLL 已锁定
31     // 0x00000100 [8]       BYPASS      (0) 将参考时钟直接传递至输出
32     // 0x0000003f [5:0]     REFDIV      (0x01) 除以PLL输入参考时钟
33     io_rw_32 cs;
34
35     _REG_(PLL_PWR_OFFSET) // PLL_PWR
36     // 控制PLL电源模式
37     // 0x00000020 [5]       VCOPD      (1) PLL VCO电源关闭 +
38     // 0x00000008 [3]       POSTDIVPD    (1) PLL 后置分频器电源关闭 +
39     // 0x00000004 [2]       DSMPD      (1) PLL DSM电源关闭 +
40     // 0x00000001 [0]       PD          (1) PLL 电源关闭 +
41     io_rw_32 pwr;
42
43     _REG_(PLL_FBDIV_INT_OFFSET) // PLL_FBDIV_INT
44     // 反馈分频器
45     // 0x00000fff [11:0]    FB DIV INT   (0x000) 详见控制寄存器说明的限制
46     io_rw_32 fbdv_int;
47
48     _REG_(PLL_PRIM_OFFSET) // PLL_PRIM
49     // 控制主输出的PLL 后置分频器
50     // 0x00070000 [18:16] POSTDIV1      (0x7) 分频系数为1至7
51     // 0x00007000 [14:12] POSTDIV2      (0x7) 分频系数为1至7
52     io_rw_32 prim;
53 } pll_hw_t;
```

SDK 定义了 `pll_init` 函数，用于配置或重新配置 PLL。该函数首先清除 PLL 中之前的电源状态，随后计算适当的反馈分频值。通过断言检查这些值是否满足上述约束条件。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pll/pll.c 第13行至21行

```

13 void pll_init(PLL pll, uint refdiv, uint vco_freq, uint post_div1, uint post_div2) {
14     uint32_t ref_freq = XOSC_HZ / refdiv;
15
16     // 检查VCO频率是否位于可接受范围内
17     assert(vco_freq >= PICO_PLL_VCO_MIN_FREQ_HZ && vco_freq <= PICO_PLL_VCO_MAX_FREQ_HZ);
18
19     // 我们将参考时钟乘以多少以获得VCO频率
20 // (寄存器命名为div，因为你将VCO输出除以该值并与参考时钟比较)
21     uint32_t fbdiv = vco_freq / ref_freq;
```

PLL的编程顺序如下：

- 编程参考时钟分频器（RP2040中为1倍分频）
- 编程反馈分频器
- 开启主电源和VCO
- 等待VCO锁定（即保持输出频率稳定）
- 设置后级分频器并启用

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/p2_common/hardware_pll/pll.c Lines 42 - 69

```

42     if ((pll->cs & PLL_CS_LOCK_BITS) &&
43         (refdiv == (pll->cs & PLL_CS_REFDIV_BITS)) &&
44         (fbdiv == (pll->fbdiv_int & PLL_FBDIV_INT_BITS)) &&
45         (pdiv == (pll->prim & (PLL_PRIM_POSTDIV1_BITS | PLL_PRIM_POSTDIV2_BITS)))) {
46         // do not disrupt PLL that is already correctly configured and operating
47         return;
48     }
49
50     reset_unreset_block_num_wait_blocking(PLL_RESET_NUM(pll));
51
52     // Load VCO-related dividers before starting VCO
53     pll->cs = refdiv;
54     pll->fbdiv_int = fbdiv;
55
56     // Turn on PLL
57     uint32_t power = PLL_PWR_PD_BITS | // Main power
58                     PLL_PWR_VCOPD_BITS; // VCO Power
59
60     hw_clear_bits(&pll->pwr, power);
61
62     // Wait for PLL to lock
63     while (!(pll->cs & PLL_CS_LOCK_BITS)) tight_loop_contents();
64
65     // Set up post dividers
66     pll->prim = pdiv;
67
68     // Turn on post divider
69     hw_clear_bits(&pll->pwr, PLL_PWR_POSTDIVPD_BITS);

```

Note the VCO is turned on first, followed by the post dividers so the PLL does not output a dirty clock while the VCO is locking.

2.18.4. List of Registers

The PLL_SYS and PLL_USB registers start at base addresses of `0x40028000` and `0x4002c000` respectively (defined as `PLL_SYS_BASE` and `PLL_USB_BASE` in SDK).

Table 274. List of PLL registers

Offset	Name	Info
0x0	CS	Control and Status
0x4	PWR	Controls the PLL power modes.
0x8	FBDIV_INT	Feedback divisor
0xc	PRIM	Controls the PLL post dividers for the primary output

PLL: CS Register

Offset: 0x0

Description

Control and Status

GENERAL CONSTRAINTS:

Reference clock frequency min=5MHz, max=800MHz

Feedback divider min=16, max=320

VCO frequency min=750MHz, max=1600MHz

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pll/pll.c 第42至69行

```

42     如果 ((pll->cs & PLL_CS_LOCK_BITS) &&
43         (refdiv == (pll->cs & PLL_CS_REFDIV_BITS)) &&
44         (fbdiv == (pll->fbdiv_int & PLL_FBDIV_INT_BITS)) &&
45         (pdiv == (pll->prim & (PLL_PRIM_POSTDIV1_BITS | PLL_PRIM_POSTDIV2_BITS))) {
46         // 不要干扰已正确配置且正在运行的PLL
47         返回;
48     }
49
50     reset_unreset_block_num_wait_blocking(PLL_RESET_NUM(pll));
51
52     // 在启动VCO之前加载与VCO相关的分频器
53     pll->cs = refdiv;
54     pll->fbdiv_int = fbdiv;
55
56     // 启动PLL
57     uint32_t power = PLL_PWR_PD_BITS | // 主电源
58                           PLL_PWR_VCOPD_BITS; // VCO电源
59
60     hw_clear_bits(&pll->pwr, power);
61
62     // 等待PLL锁定
63     while (!(pll->cs & PLL_CS_LOCK_BITS)) tight_loop_contents();
64
65     // 配置后置分频器
66     pll->prim = pdiv;
67
68     // 启用后置分频器
69     hw_clear_bits(&pll->pwr, PLL_PWR_POSTDIVPD_BITS);

```

请注意，VCO 先行启动，随后启动后置分频器，以确保 PLL 在 VCO 锁定期间不会输出杂讯时钟。

2.18.4. 寄存器列表

PLL_SYS 和 PLL_USB 寄存器的基地址分别为 `0x40028000` 和 `0x4002c000`（在 SDK 中分别定义为 `PLL_SYS_BASE` 和 `PLL_USB_BASE`）。

表 274. PLL
寄存器列表

偏移量	名称	说明
0x0	CS	控制与状态
0x4	PWR	控制 PLL 电源模式。
0x8	FBDIV_INT	反馈分频器
0xc	PRIM	控制主输出的 PLL 后置分频器

PLL：CS 寄存器

偏移量: 0x0

描述

控制与状态

通用约束:

参考时钟频率，最小值=5MHz，最大值=800MHz

反馈分频，最小值=16，最大值=320

VCO频率，最小值=750MHz，最大值=1600MHz

Table 275. CS Register

Bits	Description	Type	Reset
31	LOCK: PLL is locked	RO	0x0
30:9	Reserved.	-	-
8	BYPASS: Passes the reference clock to the output instead of the divided VCO. The VCO continues to run so the user can switch between the reference clock and the divided VCO but the output will glitch when doing so.	RW	0x0
7:6	Reserved.	-	-
5:0	REFDIV: Divides the PLL input reference clock. Behaviour is undefined for div=0. PLL output will be unpredictable during refdiv changes, wait for lock=1 before using it.	RW	0x01

PLL: PWR Register

Offset: 0x4

Description

Controls the PLL power modes.

Table 276. PWR Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5	VCOPD: PLL VCO powerdown To save power set high when PLL output not required or bypass=1.	RW	0x1
4	Reserved.	-	-
3	POSTDIVPD: PLL post divider powerdown To save power set high when PLL output not required or bypass=1.	RW	0x1
2	DSMPD: PLL DSM powerdown Nothing is achieved by setting this low.	RW	0x1
1	Reserved.	-	-
0	PD: PLL powerdown To save power set high when PLL output not required.	RW	0x1

PLL: FBDIV_INT Register

Offset: 0x8

Description

Feedback divisor
(note: this PLL does not support fractional division)

Table 277. FBDIV_INT Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:0	see ctrl reg description for constraints	RW	0x000

PLL: PRIM Register

Offset: 0xc

Description

Controls the PLL post dividers for the primary output

表275. CS寄存器

位	描述	类型	复位值
31	LOCK: PLL已锁定	只读	0x0
30:9	保留。	-	-
8	BYPASS: 直接传递参考时钟至输出，替代分频后的VCO信号。 VCO持续运行，用户可在参考时钟与分频VCO间切换，但切换时输出信号会出现毛刺。	读写	0x0
7:6	保留。	-	-
5:0	REFDIV: 对PLL输入参考时钟进行分频。 当div=0时，行为未定义。 PLL输出在refdiv更改期间不可预测，使用前请等待lock=1。	读写	0x01

PLL: PWR寄存器

偏移：0x4

说明

控制 PLL 电源模式。

表276. PWR 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	VCOPD: PLL VCO断电 在不需要PLL输出或bypass=1时，为节省功耗应置高。	读写	0x1
4	保留。	-	-
3	POSTDIVPD: PLL后级分频断电 在不需要PLL输出或bypass=1时，为节省功耗应置高。	读写	0x1
2	DSMPD: PLL DSM断电 将此设置为较低值不会产生任何效果。	读写	0x1
1	保留。	-	-
0	PD: PLL 电源关闭 当不需要 PLL 输出时，置为高电平以节省电能。	读写	0x1

PLL: FBDIV_INT 寄存器

偏移量：0x8

说明

反馈分频器

(注：该 PLL 不支持分数分频)

表 277. FBDIV_INT 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	有关限制条件，请参阅控制寄存器描述。	读写	0x000

PLL: PRIM 寄存器

偏移: 0xc

描述

控制主输出的 PLL 后置分频器

(note: this PLL does not have a secondary output)
 the primary output is driven from VCO divided by postdiv1*postdiv2

Table 278. PRIM Register

Bits	Description	Type	Reset
31:19	Reserved.	-	-
18:16	POSTDIV1: divide by 1-7	RW	0x7
15	Reserved.	-	-
14:12	POSTDIV2: divide by 1-7	RW	0x7
11:0	Reserved.	-	-

2.19. GPIO

2.19.1. Overview

RP2040 has 36 multi-functional General Purpose Input / Output (GPIO) pins, divided into two banks. In a typical use case, the pins in the QSPI bank (QSPI_SS, QSPI_SCLK and QSPI_SD0 to QSPI_SD3) are used to execute code from an external flash device, leaving the User bank (GPIO0 to GPIO29) for the programmer to use. All GPIOs support digital input and output, but GPIO26 to GPIO29 can also be used as inputs to the chip's Analogue to Digital Converter (ADC). Each GPIO can be controlled directly by software running on the processors, or by a number of other functional blocks.

The User GPIO bank supports the following functions:

- Software control via SIO (Single-Cycle IO) - [Section 2.3.1.2, "GPIO Control"](#)
- Programmable IO (PIO) - [Chapter 3, PIO](#)
- 2 × SPI - [Section 4.4, "SPI"](#)
- 2 × UART - [Section 4.2, "UART"](#)
- 2 × I2C (two-wire serial interface) - [Section 4.3, "I2C"](#)
- 8 × two-channel PWM - [Section 4.5, "PWM"](#)
- 2 × external clock inputs - [Section 2.15.2.3, "External Clocks"](#)
- 4 × general purpose clock output - [Section 2.15, "Clocks"](#)
- 4 × input to ADC - [Section 4.9, "ADC and Temperature Sensor"](#)
- USB VBUS management - [Section 4.1.2.10, "VBUS Control"](#)
- External interrupt requests, level or edge-sensitive

The QSPI bank supports the following functions:

- Software control via SIO (Single-Cycle IO) - [Section 2.3.1.2, "GPIO Control"](#)
- Flash execute in place (XIP) - [Section 2.6.3, "Flash"](#)

The logical structure of an example IO is shown in [Figure 36](#).

(注：该 PLL 不具有次级输出)
主输出由 VCO 除以 postdiv1*postdiv2 后驱动。

表 278. PRIM
寄存器

位	描述	类型	复位值
31:19	保留。	-	-
18:16	POSTDIV1 : 除以 1 至 7	读写	0x7
15	保留。	-	-
14:12	POSTDIV2 : 除以 1 至 7	读写	0x7
11:0	保留。	-	-

2.19. GPIO

2.19.1. 概述

RP2040 配备 36 个多功能通用输入输出 (GPIO) 引脚，分为两组。在典型应用中，QSPI 组的引脚（QSPI_SS、QSPI_SC_LK 及 QSPI_SD0 至 QSPI_SD3）用于从外部闪存设备执行代码，用户组（GPIO0 至 GPIO29）则供程序员使用。所有 GPIO 支持数字输入和输出，其中 GPIO26 至 GPIO29 还可作为芯片的模拟数字转换器（ADC）输入。

每个GPIO均可通过运行于处理器上的软件直接控制，或由其他若干功能模块进行控制。

用户GPIO组支持以下功能：

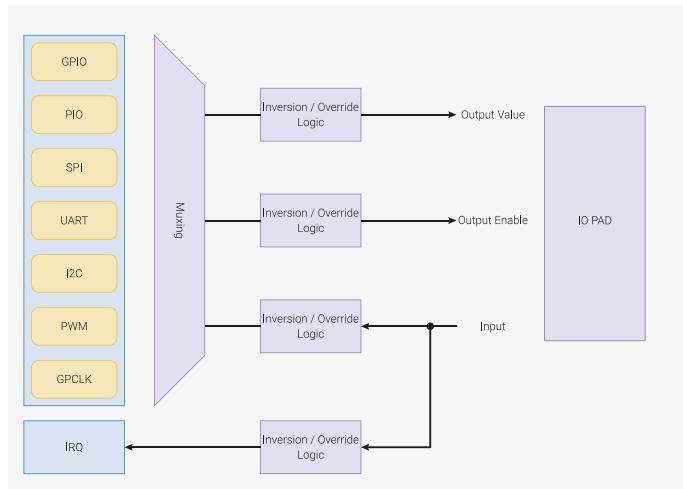
- 通过SIO（单周期IO）进行软件控制——第2.3.1.2节，“GPIO控制”
- 可编程IO（PIO）——第3章， *PIO*
- 2×SPI——第4.4节，“SPI”
- 2 ×UART——第4.2节，“UART”
- 2 ×I2C（两线串行接口）——第4.3节，“I2C”
- 8 ×双通道PWM——第4.5节，“PWM”
- 2 ×外部时钟输入——第2.15.2.3节，“外部时钟”
- 4 ×通用时钟输出——第2.15节，“时钟”
- 4 ×作为ADC输入——第4.9节，“ADC与温度传感器”
- USB VBUS管理——第4.1.2.10节，“VBUS控制”
- 外部中断请求，电平或边沿触发

QSPI 组支持以下功能：

- 通过SIO（单周期IO）进行软件控制——第2.3.1.2节，“GPIO控制”
- Flash 执行就地 (XIP) — 见第 2.6.3 节，“Flash”

示例 IO 的逻辑结构如图 36 所示。

Figure 36. Logical structure of a GPIO. Each GPIO can be controlled by one of a number of peripherals, or by software control registers in the SIO. The function select (FSEL) selects which peripheral output is in control of the GPIO's direction and output level, and/or which peripheral input can see this GPIO's input level. These three signals (output level, output enable, input level) can also be inverted, or forced high or low, using the GPIO control registers.



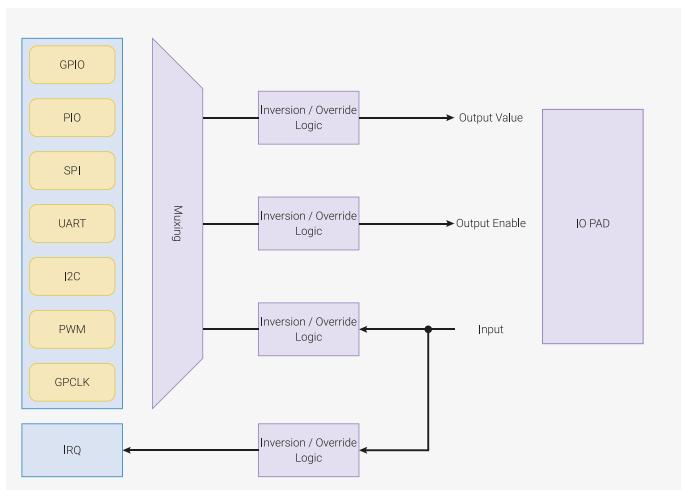
2.19.2. Function Select

The function allocated to each GPIO is selected by writing to the `FUNCSEL` field in the GPIO's `CTRL` register. See [GPIO0_CTRL](#) as an example. The functions available on each IO are shown in [Table 279](#) and [Table 281](#).

Table 279. General Purpose Input/Output (GPIO) User Bank Functions

GPIO	Function									
	F1	F2	F3	F4	F5	F6	F7	F8	F9	
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB OVCUR DET
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1			USB VBUS DET
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1			USB VBUS EN
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1			USB OVCUR DET
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1			USB VBUS DET
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1			USB VBUS EN
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1			USB OVCUR DET
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1			USB VBUS DET
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1			USB VBUS EN
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1			USB OVCUR DET
10	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1			USB VBUS DET
11	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1			USB VBUS EN
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1			USB OVCUR DET
13	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1			USB VBUS DET
14	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1			USB VBUS EN
15	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1			USB OVCUR DET
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB VBUS DET
17	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1			USB VBUS EN
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1			USB OVCUR DET
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1			USB VBUS DET
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0		USB VBUS EN

图36。GPIO的逻辑结构。
每个GPIO可以由多个外设之一控制，或由SIO中的软件控制寄存器控制。
功能选择(FSEL)用于选择由哪个外设输出控制GPIO的方向和输出电平，和／或由哪个外设输入读取该GPIO的输入电平。这三个信号（输出电平、输出使能、输入电平）也可以通过GPIO控制寄存器进行反相，或强制置高或置低。



2.19.2. 功能选择

分配给每个GPIO的功能，通过向GPIO的CTRL寄存器中的FUNCSEL字段写入来选择。以GPIO0_CTRL为例。各IO上可用的功能详见表 279 和表 281。

表279。通用输入/输出(GPIO)用户组功能

GPIO	功能									
	F1	F2	F3	F4	F5	F6	F7	F8	F9	
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB 过流检测
1	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1			USB 总线电压检测
2	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
3	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1			USB 过流检测
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1			USB 总线电压检测
5	SPI0 片选信号 (CSn)	UART1 RX	I2C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
6	SPI0 时钟信号 (SCK)	UART1 CTS	I2C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1			USB 过流检测
7	SPI0 发送数据 (TX)	UART1 RTS	I2C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1			USB 总线电压检测
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
9	SPI1 CSn	UART1 RX	I2C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1			USB 过流检测
10	SPI1 SCK	UART1 CTS	I2C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1			USB 总线电压检测
11	SPI1 TX	UART1 RTS	I2C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1			USB 过流检测
13	SPI1 CSn	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1			USB 总线电压检测
14	SPI1 SCK	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM7 A	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
15	SPI1 TX	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM7 B	SIO	PIO0	PIO1			USB 过流检测
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1			USB 总线电压检测
17	SPI0 片选信号 (CSn)	UART0 接收 (RX)	I2C0 时钟线 (SCL)	PWM0 通道B	SIO	PIO0	PIO1			USB 总线使能 (VBUS EN)
18	SPI0 时钟信号 (SCK)	UART0 清除发送 (CTS)	I2C1 数据线 (SDA)	PWM1 通道A	SIO	PIO0	PIO1			USB 过流检测
19	SPI0 发送数据 (TX)	UART0 请求发送 (RTS)	I2C1 时钟线 (SCL)	PWM1 通道B	SIO	PIO0	PIO1			USB 总线电压检测
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIO0		USB 总线使能 (VBUS EN)

	Function									
21	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB OVCUR DET	
22	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB VBUS DET	
23	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB VBUS EN	
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB OVCUR DET	
25	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB VBUS DET	
26	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS EN	
27	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB OVCUR DET	
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB VBUS DET	
29	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS EN	

Each GPIO can have one function selected at a time. Likewise, each peripheral input (e.g. UART0 RX) should only be selected on one GPIO at a time. If the same peripheral input is connected to multiple GPIOs, the peripheral sees the logical OR of these GPIO inputs.

Table 280. GPIO User Bank function descriptions

Function Name	Description
SPIx	Connect one of the internal PL022 SPI peripherals to GPIO
UARTx	Connect one of the internal PL011 UART peripherals to GPIO
I2Cx	Connect one of the internal DW I2C peripherals to GPIO
PWMx A/B	Connect a PWM slice to GPIO. There are eight PWM slices, each with two output channels (A/B). The B pin can also be used as an input, for frequency and duty cycle measurement.
SIO	Software control of GPIO, from the single-cycle IO (SIO) block. The SIO function (F5) must be selected for the processors to <i>drive</i> a GPIO, but the input is always connected, so software can check the state of GPIOs at any time.
PIOx	Connect one of the programmable IO blocks (PIO) to GPIO. PIO can implement a <i>wide</i> variety of interfaces, and has its own internal pin mapping hardware, allowing flexible placement of digital interfaces on user bank GPIOs. The PIO function (F6, F7) must be selected for PIO to <i>drive</i> a GPIO, but the input is always connected, so the PIOs can always see the state of all pins.
CLOCK GPINx	General purpose clock inputs. Can be routed to a number of internal clock domains on RP2040, e.g. to provide a 1Hz clock for the RTC, or can be connected to an internal frequency counter.
CLOCK GPOUTx	General purpose clock outputs. Can drive a number of internal clocks onto GPIOs, with optional integer divide.
USB OVCUR DET/VBUS DET/VBUS EN	USB power control signals to/from the internal USB controller

Table 281. General Purpose Input/Output (GPIO) QSPI Bank Functions

IO	Function									
F0		F1	F2	F3	F4	F5	F6	F7	F8	F9
QSPI SCK	XIP SCK					SIO				
QSPI CSn	XIP CSn					SIO				
QSPI SD0	XIP SD0					SIO				

功能									
21	SPI0 片选信号 (CSn)	UART1 RX	I ₂ C0 时钟线 (SCL)	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB 过流检测
22	SPI0 时钟信号 (SCK)	UART1 CTS	I ₂ C1 数据线 (SDA)	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB 总线电压检测
23	SPI0 发送数据 (TX)	UART1 RTS	I ₂ C1 时钟线 (SCL)	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB 总线使能 (VBUS EN)
24	SPI1 RX	UART1 TX	I ₂ C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB 过流检测
25	SPI1 CSn	UART1 RX	I ₂ C0 时钟线 (SCL)	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB 总线电压检测
26	SPI1 SCK	UART1 CTS	I ₂ C1 数据线 (SDA)	PWM5 A	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)
27	SPI1 TX	UART1 RTS	I ₂ C1 时钟线 (SCL)	PWM5 B	SIO	PIO0	PIO1		USB 过流检测
28	SPI1 RX	UART0 TX	I ₂ C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB 总线电压检测
29	SPI1 CSn	UART0 接收 (RX)	I ₂ C0 时钟线 (SCL)	PWM6 B	SIO	PIO0	PIO1		USB 总线使能 (VBUS EN)

每个GPIO一次仅能选择一个功能。同样，每个外设输入（例如UART0 RX）一次应仅选择一个GPIO。若同一外设输入连接至多个GPIO，该外设将接收到这些GPIO输入的逻辑“或”值。

表200。 GPIO用户组功能
描述

功能名称	描述
SPIx	将内部 PL022 SPI 外设之一连接至 GPIO
UARTx	将内部 PL011 UART 外设之一连接至 GPIO
I ₂ Cx	将内部 DW I ₂ C 外设之一连接至 GPIO
PWMx A/B	将PWM切片连接至GPIO。共有八个PWM切片，每个切片包含两个输出通道（A/B）。B引脚亦可用作输入，用于频率和占空比的测量。
SIO	通过单周期IO (SIO) 模块实现GPIO的软件控制。必须选择SIO功能 (F5) 以使处理器驱动GPIO，但输入端始终连接，因此软件能够随时检测GPIO状态。
PIOx	将其中一个可编程IO块 (PIO) 连接至GPIO。PIO可实现多种接口，并具备自身的内部引脚映射硬件，允许用户组GPIO上灵活布置数字接口。必须选择PIO功能 (F6, F7) 以使PIO驱动GPIO，但输入始终连接，因此PIO始终能够读取所有引脚状态。
CLOCK GPINx	通用时钟输入。可路由至RP2040的多个内部时钟域，例如为RTC提供1Hz时钟，或连接至内部频率计数器。
CLOCK GPOUTx	通用时钟输出。可驱动多个内部时钟至GPIO，并支持可选的整数分频。
USB OVCUR DET/VBUS DET/VBUS EN	内部USB控制器的USB电源控制信号。

表201。通用输入/输出 (GPIO) QSPI 银行功能

功能	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9
QSPI SCK	XIP SCK					SIO				
QSPI CSn	XIP CSn					SIO				
QSPI SD0	XIP SD0					SIO				

	Function									
QSPI SD1	XIP SD1					SIO				
QSPI SD2	XIP SD2					SIO				
QSPI SD3	XIP SD3					SIO				

Table 282. GPIO QSPI Bank function descriptions

Function Name	Description
XIP	Connection to the synchronous serial interface (SSI) inside the flash execute in place (XIP) subsystem. This allows processors to execute code directly from an external SPI, Dual-SPI or Quad-SPI flash
SIO	Software control of GPIO, from the single-cycle IO (SIO) block. The SIO function (F5) must be selected for the processors to drive a GPIO, but the input is always connected, so software can check the state of GPIOs at any time. The QSPI IOs are controlled via the <code>SIO_GPIO_HI_x</code> registers, and are mapped to register bits in the order SCK, CSn, SD0, SD1, SD2, SD3, starting at the LSB.

The six QSPI Bank GPIO pins are typically used by the XIP peripheral to communicate with an external flash device. However, there are two scenarios where the pins can be used as software-controlled GPIOs:

- If a SPI or Dual-SPI flash device is used for execute-in-place, then the SD2 and SD3 pins are not used for flash access, and can be used for other GPIO functions on the circuit board.
- If RP2040 is used in a flashless configuration (USB boot only), then all six pins can be used for software-controlled GPIO functions

2.19.3. Interrupts

An interrupt can be generated for every GPIO pin in four scenarios:

- Level High: the GPIO pin is a logical 1
- Level Low: the GPIO pin is a logical 0
- Edge High: the GPIO has transitioned from a logical 0 to a logical 1
- Edge Low: the GPIO has transitioned from a logical 1 to a logical 0

The level interrupts are not latched. This means that if the pin is a logical 1 and the level high interrupt is active, it will become inactive as soon as the pin changes to a logical 0. The edge interrupts are stored in the `INTR` register and can be cleared by writing to the `INTR` register.

There are enable, status, and force registers for three interrupt destinations: proc 0, proc 1, and dormant_wake. For proc 0 the registers are enable (`PROCO_INTE0`), status (`PROCO_INTS0`), and force (`PROCO_INTF0`). Dormant wake is used to wake the ROSC or XOSC up from dormant mode. See [Section 2.11.5.2](#) for more information on dormant mode.

All interrupts are ORed together per-bank per-destination resulting in a total of six GPIO interrupts:

- IO bank 0 to dormant wake
- IO bank 0 to proc 0
- IO bank 0 to proc 1
- IO QSPI to dormant wake
- IO QSPI to proc 0
- IO QSPI to proc 1

This means the user can watch for several GPIO events at once.

	功能									
QSPI SD1	XIP SD1					SIO				
QSPI SD2	XIP SD2					SIO				
QSPI SD3	XIP SD3					SIO				

表 282 GPIO QSPI
银行功能
说明

功能名称	描述
XIP	连接至闪存就地执行（XIP）子系统内的同步串行接口（SSI）。此功能使处理器能够直接从外部 SPI、双 SPI 或四线 SPI 闪存中执行代码。
SIO	通过单周期IO（SIO）模块实现GPIO的软件控制。必须选择SIO功能（F5）以驱动GPIO，但输入始终连接，因此软件可随时检测GPIO状态。QSPI IO通过SIO_GPIO_HI_x寄存器控制，且按SCK、CSn、SD0、SD1、SD2、SD3的顺序映射到寄存器位，从最低有效位开始。

这六个QSPI Bank GPIO引脚通常由XIP外设用于与外部闪存设备通信。

然而，存在两种情况下，这些引脚可用作软件控制的GPIO：

- 若采用SPI或Dual-SPI闪存设备执行就地（execute-in-place），则SD2和SD3引脚不用于闪存访问，可用于电路板上的其他GPIO功能。
- 若RP2040采用无闪存配置（仅支持USB启动），则所有六个引脚均可用作软件控制的GPIO功能。

2.19.3. 中断

每个GPIO引脚在以下四种情形下均可触发中断：

- 高电平：GPIO引脚处于逻辑1状态
- 低电平：GPIO引脚处于逻辑0状态
- 上升沿：GPIO从逻辑0跳变至逻辑1
- 下降沿：GPIO从逻辑1跳变至逻辑0

电平中断不会被锁存。这意味着，当引脚为逻辑1且高电平中断处于激活状态时，一旦引脚变为逻辑0，该中断将立即失效。边缘中断的信息存储于 INTR 寄存器，可通过向 INTR 寄存器写入数据予以清除。

针对三类中断目标：proc 0、proc 1 和 dormant_wake，分别设有使能、状态及强制寄存器。针对 proc 0，相关寄存器分别为使能（PROC0_INTE0）、状态（PROC0_INTS0）及强制（PROC0_INTFO）。休眠唤醒用于将 ROSC 或 XOSC 从休眠模式中唤醒。有关休眠模式的详细信息，请参见第2.11.5.2节。

所有中断均在每个总线及每个目标处进行逻辑或运算，合计生成六个GPIO中断：

- IO总线0至休眠唤醒
- IO总线0至处理器0
- IO总线0至处理器1
- IO QSPI至休眠唤醒
- IO QSPI至处理器0
- IO QSPI至处理器1

这意味着用户可以同时监控多个GPIO事件。

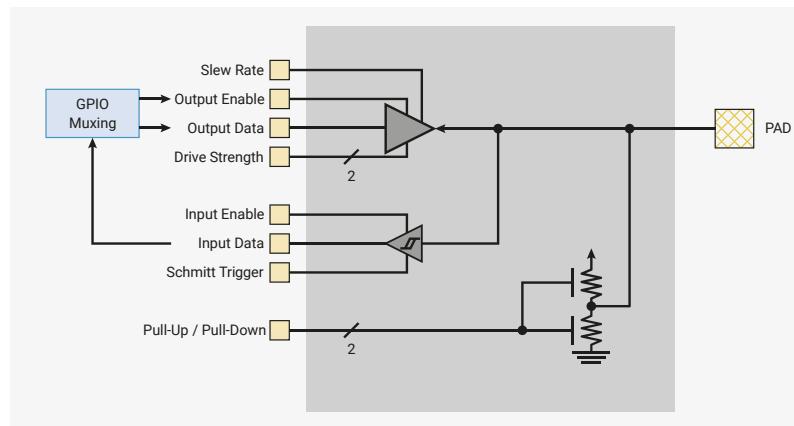
2.19.4. Pads

Each GPIO is connected to the off-chip world via a "pad". Pads are the electrical interface between the chip's internal logic and external circuitry. They translate signal voltage levels, support higher currents and offer some protection against electrostatic discharge (ESD) events. Pad electrical behaviour can be adjusted to meet the requirements of the external circuitry. The following adjustments are available:

- Output drive strength can be set to 2mA, 4mA, 8mA or 12mA
- Output slew rate can be set to slow or fast
- Input hysteresis (schmitt trigger mode) can be enabled
- A pull-up or pull-down can be enabled, to set the output signal level when the output driver is disabled
- The input buffer can be disabled, to reduce current consumption when the pad is unused, unconnected or connected to an analogue signal.

An example pad is shown in [Figure 37](#).

Figure 37. Diagram of a single IO pad.



The pad's Output Enable, Output Data and Input Data ports are connected, via the IO mux, to the function controlling the pad. All other ports are controlled from the pad control register. The register also allows the pad's output driver to be disabled, by overriding the Output Enable signal from the function controlling the pad. See [GPIO0](#) for an example of a pad control register.

Both the output signal level and acceptable input signal level at the pad are determined by the digital IO supply (IOVDD). IOVDD can be any nominal voltage between 1.8V and 3.3V, but to meet specification when powered at 1.8V, the pad input thresholds must be adjusted by writing a 1 to the pad [VOLTAGE_SELECT](#) registers. By default the pad input thresholds are valid for an IOVDD voltage between 2.5V and 3.3V. Using a voltage of 1.8V with the default input thresholds is a safe operating mode, though it will result in input thresholds that don't meet specification.

WARNING

Using IOVDD voltages greater than 1.8V, with the input thresholds set for 1.8V may result in damage to the chip.

Pad input threshold are adjusted on a per bank basis, with separate [VOLTAGE_SELECT](#) registers for the pads associated with the User IO bank (IO Bank 0) and the QSPI IO bank. However, both banks share the same digital IO supply (IOVDD), so both register should always be set to the same value.

Pad register details are available in [Section 2.19.6.3, "Pad Control - User Bank"](#) and [Section 2.19.6.4, "Pad Control - QSPI Bank"](#).

2.19.4.1. Bus Keeper Mode

For each pad, only the pull-up or the pull-down resistor can be enabled at any given time. It is impossible to enable both simultaneously. Instead, if you set both the [GPIO0.PDE](#) and [GPIO0.PUE](#) bits simultaneously then you enable **bus keeper**

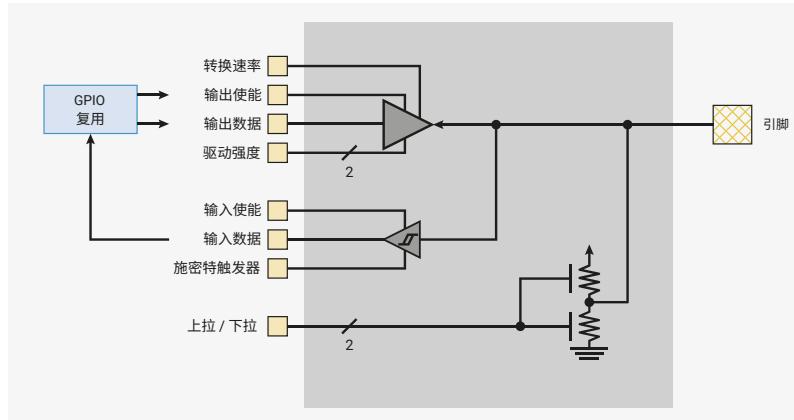
2.19.4. 引脚

每个GPIO通过“引脚”连接到芯片外部世界。引脚是芯片内部逻辑与外部电路之间的电气接口。引脚转换信号电压水平，支持更高电流，并为静电放电（ESD）事件提供一定的防护。引脚的电气特性可调节，以满足外部电路的需求。提供以下调整选项：

- 输出驱动强度可设置为 2mA、4mA、8mA 或 12mA
- 输出转换速率可设置为慢速或快速
- 可启用输入迟滞（施密特触发器模式）
- 可启用上拉或下拉，以在输出驱动器禁用时设定输出信号电平
- 输入缓冲区可被禁用，以降低在引脚未使用、未连接或连接至模拟信号时的电流消耗。

图 37 展示了一个示例引脚。

图 37。单个 IO 引脚的示意图



该引脚的输出使能、输出数据和输入数据端口通过 IO 复用连接至控制该引脚功能。所有其他端口由引脚控制寄存器管理。该寄存器还允许通过覆盖控制该引脚功能的输出使能信号来禁用该引脚的输出驱动器。有关引脚控制寄存器的示例，请参见 GPIO0。

输出信号电平及垫片的可接受输入信号电平均由数字IO电源（IOVDD）决定。

IOVDD可为1.8V至3.3V之间的任意标称电压，但为满足1.8V供电时的规范，必须通过向垫片VOLTAGE_SELECT寄存器写入1以调整垫片输入阈值。默认情况下，垫片输入阈值适用于IOVDD电压在2.5V至3.3V之间。使用1.8V电压且保持默认输入阈值为安全操作模式，但输入阈值将不符合规格要求。

警告

当输入阈值设为1.8V时，使用高于1.8V的IOVDD电压可能会对芯片造成损坏。

垫片输入阈值按银行分别调整，对用户IO银行（IO Bank 0）和QSPI IO银行的垫片分别设有各自的VOLTAGE_SELECT寄存器。然而，两个寄存器组共享相同的数字IO电源（IOVDD），因此两个寄存器应始终设置为相同的值。

寄存器详细信息详见第2.19.6.3节“引脚控制 - 用户寄存器组”和第2.19.6.4节“引脚控制 - QSPI 寄存器组”。

2.19.4.1. 总线保持模式

对于每个引脚，在任何时刻只能启用上拉电阻或下拉电阻其中之一。不可能同时启用两者。相反，若同时设置GPIO0.PDE 和GPIO0.PUE位，则启用总线保持

mode, where the pad is:

- pulled up when its input is high, and
- pulled down when its input is low

When the output buffer is disabled, and the pad is not driven by any external source, this mode weakly retains the pad's current logical state. The pad does not float to mid-rail.

2.19.5. Software Examples

2.19.5.1. Select an IO function

An IO pin can perform many different functions and must be configured before use. For example, you may want it to be a `UART_TX` pin, or a `PWM` output. The SDK provides `gpio_set_function` for this purpose. Many SDK examples will call `gpio_set_function` at the beginning so that it can print to a UART.

The SDK starts by defining a structure to represent the registers of IO bank 0, the User IO bank. Each IO has a status register, followed by a control register. There are 30 IOs, so the structure containing a status and control register is instantiated as `io[30]` to repeat it 30 times.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/io_bank0.h Lines 181 - 229

```

181 typedef struct {
182     io_bank0_status_ctrl_hw_t io[30];
183
184     // (Description copied from array index 0 register IO_BANK0_INTR0 applies similarly to other array indexes)
185     _REG_(IO_BANK0_INTR0_OFFSET) // IO_BANK0_INTR0
186     // Raw Interrupts
187     // 0x80000000 [31]    GPIO7_EDGE_HIGH (0)
188     // 0x40000000 [30]    GPIO7_EDGE_LOW (0)
189     // 0x20000000 [29]    GPIO7_LEVEL_HIGH (0)
190     // 0x10000000 [28]    GPIO7_LEVEL_LOW (0)
191     // 0x08000000 [27]    GPIO6_EDGE_HIGH (0)
192     // 0x04000000 [26]    GPIO6_EDGE_LOW (0)
193     // 0x02000000 [25]    GPIO6_LEVEL_HIGH (0)
194     // 0x01000000 [24]    GPIO6_LEVEL_LOW (0)
195     // 0x00800000 [23]    GPIO5_EDGE_HIGH (0)
196     // 0x00400000 [22]    GPIO5_EDGE_LOW (0)
197     // 0x00200000 [21]    GPIO5_LEVEL_HIGH (0)
198     // 0x00100000 [20]    GPIO5_LEVEL_LOW (0)
199     // 0x00080000 [19]    GPIO4_EDGE_HIGH (0)
200     // 0x00040000 [18]    GPIO4_EDGE_LOW (0)
201     // 0x00020000 [17]    GPIO4_LEVEL_HIGH (0)
202     // 0x00010000 [16]    GPIO4_LEVEL_LOW (0)
203     // 0x00008000 [15]    GPIO3_EDGE_HIGH (0)
204     // 0x00004000 [14]    GPIO3_EDGE_LOW (0)
205     // 0x00002000 [13]    GPIO3_LEVEL_HIGH (0)
206     // 0x00001000 [12]    GPIO3_LEVEL_LOW (0)
207     // 0x00000800 [11]    GPIO2_EDGE_HIGH (0)
208     // 0x00000400 [10]    GPIO2_EDGE_LOW (0)
209     // 0x00000200 [9]     GPIO2_LEVEL_HIGH (0)
210     // 0x00000100 [8]     GPIO2_LEVEL_LOW (0)
211     // 0x00000080 [7]     GPIO1_EDGE_HIGH (0)
212     // 0x00000040 [6]     GPIO1_EDGE_LOW (0)
213     // 0x00000020 [5]     GPIO1_LEVEL_HIGH (0)
214     // 0x00000010 [4]     GPIO1_LEVEL_LOW (0)
215     // 0x00000008 [3]     GPIO0_EDGE_HIGH (0)
216     // 0x00000004 [2]     GPIO0_EDGE_LOW (0)

```

模式，在该模式下引脚：

- 当输入为高电平时被上拉，
- 当输入为低电平时被下拉。

当输出缓冲器被禁用且引脚未被任何外部信号驱动时，此模式能够弱保持引脚当前的逻辑状态。引脚不会漂浮至中间电平。

2.19.5. 软件示例

2.19.5.1. 选择一个IO功能

一个IO引脚可以执行多种不同功能，必须在使用前进行配置。例如，您可能希望它作为 `UART_TX` 引脚，或作为 `PWM` 输出。SDK提供了`gpio_set_function`函数以实现此目的。许多SDK示例会在初始化时调用`gpio_set_function`，以便通过UART进行打印。

SDK首先定义了一个结构体，用于表示IO银行0，即用户IO银行的寄存器。每个IO均包含一个状态寄存器，紧接其后的是一个控制寄存器。共有30个IO，因此包含状态和控制寄存器的结构体实例被定义为 `io[30]`，实现了30次重复。

SDK链接：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/io_bank0.h 第181行至第229行

```

181 typedef struct {
182     io_bank0_status_ctrl_hw_t io[30];
183
184 // (描述摘自数组索引0寄存器IO_BANK0_INTR0，其他数组索引适用同样规则)
185     _REG_(IO_BANK0_INTR0_OFFSET) // IO_BANK0_INTR0
186     // 原始中断
187     // 0x80000000 [31]      GPIO7_EDGE_HIGH (0)
188     // 0x40000000 [30]      GPIO7_EDGE_LOW (0)
189     // 0x20000000 [29]      GPIO7_LEVEL_HIGH (0)
190     // 0x10000000 [28]      GPIO7_LEVEL_LOW (0)
191     // 0x08000000 [27]      GPIO6_EDGE_HIGH (0)
192     // 0x04000000 [26]      GPIO6_EDGE_LOW (0)
193     // 0x02000000 [25]      GPIO6_LEVEL_HIGH (0)
194     // 0x01000000 [24]      GPIO6_LEVEL_LOW (0)
195     // 0x00800000 [23]      GPIO5_EDGE_HIGH (0)
196     // 0x00400000 [22]      GPIO5_EDGE_LOW (0)
197     // 0x00200000 [21]      GPIO5_LEVEL_HIGH (0)
198     // 0x00100000 [20]      GPIO5_LEVEL_LOW (0)
199     // 0x00080000 [19]      GPIO4_EDGE_HIGH (0)
200     // 0x00040000 [18]      GPIO4_EDGE_LOW (0)
201     // 0x00020000 [17]      GPIO4_LEVEL_HIGH (0)
202     // 0x00010000 [16]      GPIO4_LEVEL_LOW (0)
203     // 0x00008000 [15]      GPIO3_EDGE_HIGH (0)
204     // 0x00004000 [14]      GPIO3_EDGE_LOW (0)
205     // 0x00002000 [13]      GPIO3_LEVEL_HIGH (0)
206     // 0x00001000 [12]      GPIO3_LEVEL_LOW (0)
207     // 0x00000800 [11]      GPIO2_EDGE_HIGH (0)
208     // 0x00000400 [10]      GPIO2_EDGE_LOW (0)
209     // 0x00000200 [9]       GPIO2_LEVEL_HIGH (0)
210     // 0x00000100 [8]       GPIO2_LEVEL_LOW (0)
211     // 0x00000080 [7]       GPIO1_EDGE_HIGH (0)
212     // 0x00000040 [6]       GPIO1_EDGE_LOW (0)
213     // 0x00000020 [5]       GPIO1_LEVEL_HIGH (0)
214     // 0x00000010 [4]       GPIO1_LEVEL_LOW (0)
215     // 0x00000008 [3]       GPIO0_EDGE_HIGH (0)
216     // 0x00000004 [2]       GPIO0_EDGE_LOW (0)

```

```

217     // 0x00000002 [1]      GPIO00_LEVEL_HIGH (0)
218     // 0x00000001 [0]      GPIO00_LEVEL_LOW (0)
219     io_rw_32 intr[4];
220
221     union {
222         struct {
223             io_bank0_irq_ctrl_hw_t proc0_irq_ctrl;
224             io_bank0_irq_ctrl_hw_t proc1_irq_ctrl;
225             io_bank0_irq_ctrl_hw_t dormant_wake_irq_ctrl;
226         };
227         io_bank0_irq_ctrl_hw_t irq_ctrl[3];
228     };
229 } io_bank0_hw_t;

```

A similar structure is defined for the pad control registers for IO bank 1. By default, all pads come out of reset ready to use, with their input enabled and output disable set to 0. Regardless, `gpio_set_function` in the SDK sets these to make sure the pad is ready to use by the selected function. Finally, the desired function select is written to the IO control register (see `GPIO0_CTRL` for an example of an IO control register).

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c Lines 36 - 53

```

36 // Select function for this GPIO, and ensure input/output are enabled at the pad.
37 // This also clears the input/output/irq override bits.
38 void gpio_set_function(uint gpio, gpio_function_t fn) {
39     check_gpio_param(gpio);
40     invalid_params_if(HARDWARE_GPIO, ((uint32_t)fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB) &
41         ~IO_BANK0_GPIO0_CTRL_FUNCSEL_BITS);
42     // Set input enable on, output disable off
43     hw_write_masked(&pads_bank0_hw->io[gpio],
44                     PADS_BANK0_GPIO0_IE_BITS,
45                     PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
46     );
47     // Zero all fields apart from fsel; we want this IO to do what the peripheral tells it.
48     // This doesn't affect e.g. pullup/pulldown, as these are in pad controls.
49     io_bank0_hw->io[gpio].ctrl = fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
50 }

```

2.19.5.2. Enable a GPIO interrupt

The SDK provides a method of being interrupted when a GPIO pin changes state:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c Lines 186 - 196

```

186 void gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled) {
187     // either this call disables the interrupt or callback should already be set.
188     // this protects against enabling the interrupt without callback set
189     assert(!enabled || irq_has_handler(IO_IRQ_BANK0));
190
191     // Separate mask/force/status per-core, so check which core called, and
192     // set the relevant IRQ controls.
193     io_bank0_irq_ctrl_hw_t *irq_ctrl_base = get_core_num() ?
194         &io_bank0_hw->proc1_irq_ctrl : &io_bank0_hw-
195         >proc0_irq_ctrl;
196     _gpio_set_irq_enabled(gpio, events, enabled, irq_ctrl_base);
197 }

```

`gpio_set_irq_enabled` uses a lower level function `_gpio_set_irq_enabled`:

```

217     // 0x00000002 [1]      GPIO00_LEVEL_HIGH (0)
218     // 0x00000001 [0]      GPIO00_LEVEL_LOW (0)
219     io_rw_32_intr[4];
220
221     union {
222         struct {
223             io_bank0_irq_ctrl_hw_t proc0_irq_ctrl;
224             io_bank0_irq_ctrl_hw_t proc1_irq_ctrl;
225             io_bank0_irq_ctrl_hw_t dormant_wake_irq_ctrl;
226         };
227         io_bank0_irq_ctrl_hw_t irq_ctrl[3];
228     };
229 } io_bank0_hw_t;

```

为 IO bank 1 的引脚控制寄存器定义了类似结构。默认情况下，所有引脚解除复位后均可直接使用，输入启用，输出禁用设为 0。无论如何，SDK 中的 `gpio_set_function` 函数会设置这些寄存器，以确保引脚可用于所选功能。最后，将所需功能选择写入 IO 控制寄存器（参见 `GPIO0_CTRL` 以获取 IO 控制寄存器示例）。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 36 至 53 行

```

36 // 为此 GPIO 选择功能，并确保引脚输入输出已启用。
37 // 此操作还会清除输入、输出及中断覆盖位。
38 void gpio_set_function(uint gpio, gpio_function_t fn) {
39     check_gpio_param(gpio);
40     invalid_params_if(HARDWARE_GPIO, ((uint32_t)fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB) &
41         ~IO_BANK0_GPIO0_CTRL_FUNCSEL_BITS);
42     // 设置输入使能为开启，输出使能为关闭
43     hw_write_masked(&pads_bank0_hw->io[gpio],
44                     PADS_BANK0_GPIO0_IE_BITS,
45                     PADS_BANK0_GPIO0_IE_BITS | PADS_BANK0_GPIO0_OD_BITS
46     );
47     // 将除 fsel 以外的所有字段清零；我们希望该 IO 按照外设指示执行。
48     // 这不会影响例如上拉/下拉，因为这些设置位于引脚控制寄存器中。
49     io_bank0_hw->io[gpio].ctrl = fn << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
50 }

```

2.19.5.2. 启用 GPIO 中断

SDK 提供了当 GPIO 引脚状态变化时触发中断的方法：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 186-196 行

```

186 void gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled) {
187     // 此调用要么禁用中断，要么回调函数应已设置。
188     // 该机制防止在未设置回调函数时启用中断。
189     assert(!enabled || irq_has_handler(IO_IRQ_BANK0));
190
191     // 每个核心维护独立的掩码/强制/状态，因此需检查调用的是哪个核心，
192     // 并设置对应的IRQ控制。
193     io_bank0_irq_ctrl_hw_t *irq_ctrl_base = get_core_num() ?
194         &io_bank0_hw->proc1_irq_ctrl : &io_bank0_hw-
195         >proc0_irq_ctrl;
196     _gpio_set_irq_enabled(gpio, events, enabled, irq_ctrl_base);
197 }

```

`gpio_set_irq_enabled` 调用了更底层函数 `_gpio_set_irq_enabled`：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c Lines 173 - 184

```

173 static void _gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled,
174     io_bank0_irq_ctrl_hw_t *irq_ctrl_base) {
175     // Clear stale events which might cause immediate spurious handler entry
176     gpio_acknowledge_irq(gpio, events);
177 
178     io_rw_32 *en_reg = &irq_ctrl_base->inte[	gpio / 8];
179     events <= 4 * (gpio % 8);
180 
181     if (enabled)
182         hw_set_bits(en_reg, events);
183     else
184         hw_clear_bits(en_reg, events);
185 }
```

The user provides a pointer to a callback function that is called when the GPIO event happens. An example application that uses this system is `hello_gpio_irq`:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_gpio_irq/hello_gpio_irq.c

```

1 /**
2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3  *
4  * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8 #include "pico/stl.h"
9 #include "hardware/gpio.h"
10
11 #define GPIO_WATCH_PIN 2
12
13 static char event_str[128];
14
15 void gpio_event_string(char *buf, uint32_t events);
16
17 void gpio_callback(uint gpio, uint32_t events) {
18     // Put the GPIO event(s) that just happened into event_str
19     // so we can print it
20     gpio_event_string(event_str, events);
21     printf("GPIO %d %s\n", gpio, event_str);
22 }
23
24 int main() {
25     stdio_init_all();
26
27     printf("Hello GPIO IRQ\n");
28     gpio_init(GPIO_WATCH_PIN);
29     gpio_set_irq_enabled_with_callback(GPIO_WATCH_PIN, GPIO_IRQ_EDGE_RISE |
30                                         GPIO_IRQ_EDGE_FALL, true, &gpio_callback);
31
32     // Wait forever
33     while (1);
34 }
35
36 static const char *gpio_irq_str[] = {
37     "LEVEL_LOW", // 0x1
38     "LEVEL_HIGH", // 0x2
39     "EDGE_FALL", // 0x4
40     "EDGE_RISE" // 0x8
41 }
```

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_gpio/gpio.c 第 173 至 184 行

```

173 static void _gpio_set_irq_enabled(uint gpio, uint32_t events, bool enabled,
174     io_bank0_irq_ctrl_hw_t *irq_ctrl_base) {
175     // 清除可能导致立即虚假中断处理程序执行的陈旧事件
176     gpio_acknowledge_irq(gpio, events);
177 
178     io_rw_32 *en_reg = &irq_ctrl_base->inte[	gpio / 8];
179     events <= 4 * (gpio % 8);
180 
181     if (enabled)
182         hw_set_bits(en_reg, events);
183     else
184         hw_clear_bits(en_reg, events);
185 }
```

用户应提供指向回调函数的指针，当 GPIO 事件发生时将调用该函数。使用该系统的示例应用为 hello_gpio_irq：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_gpio_irq/hello_gpio_irq.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX 许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8 #include "pico/stl.h"
9 #include "hardware/gpio.h"
10
11 #define GPIO_WATCH_PIN 2
12
13 static char event_str[128];
14
15 void gpio_event_string(char *buf, uint32_t events);
16
17 void gpio_callback(uint gpio, uint32_t events) {
18     // 将刚发生的 GPIO 事件存入 event_str
19     // 以便打印
20     gpio_event_string(event_str, events);
21     printf("GPIO %d %s\n", gpio, event_str);
22 }
23
24 int main() {
25     stdio_init_all();
26
27     printf("Hello GPIO IRQ\n");
28     gpio_init(GPIO_WATCH_PIN);
29     gpio_set_irq_enabled_with_callback(GPIO_WATCH_PIN, GPIO_IRQ_EDGE_RISE |
30         GPIO_IRQ_EDGE_FALL, true, &gpio_callback);
31
32     // 永久等待
33     while (1);
34
35
36 static const char *gpio_irq_str[] = {
37     "LEVEL_LOW", // 0x1
38     "LEVEL_HIGH", // 0x2
39     "EDGE_FALL", // 0x4
40     "EDGE_RISE" // 0x8
41 }
```

```

41 };
42
43 void gpio_event_string(char *buf, uint32_t events) {
44     for (uint i = 0; i < 4; i++) {
45         uint mask = (1 << i);
46         if (events & mask) {
47             // Copy this event string into the user string
48             const char *event_str = gpio_irq_str[i];
49             while (*event_str != '\0') {
50                 *buf++ = *event_str++;
51             }
52             events &= ~mask;
53
54             // If more events add ","
55             if (events) {
56                 *buf++ = ',';
57                 *buf++ = ' ';
58             }
59         }
60     }
61     *buf++ = '\0';
62 }

```

2.19.6. List of Registers

2.19.6.1. IO - User Bank

The User Bank IO registers start at a base address of [0x40014000](#) (defined as [IO_BANK0_BASE](#) in SDK).

Table 283. List of
IO_BANK0 registers

Offset	Name	Info
0x000	GPIO0_STATUS	GPIO status
0x004	GPIO0_CTRL	GPIO control including function select and overrides.
0x008	GPIO1_STATUS	GPIO status
0x00c	GPIO1_CTRL	GPIO control including function select and overrides.
0x010	GPIO2_STATUS	GPIO status
0x014	GPIO2_CTRL	GPIO control including function select and overrides.
0x018	GPIO3_STATUS	GPIO status
0x01c	GPIO3_CTRL	GPIO control including function select and overrides.
0x020	GPIO4_STATUS	GPIO status
0x024	GPIO4_CTRL	GPIO control including function select and overrides.
0x028	GPIO5_STATUS	GPIO status
0x02c	GPIO5_CTRL	GPIO control including function select and overrides.
0x030	GPIO6_STATUS	GPIO status
0x034	GPIO6_CTRL	GPIO control including function select and overrides.
0x038	GPIO7_STATUS	GPIO status
0x03c	GPIO7_CTRL	GPIO control including function select and overrides.
0x040	GPIO8_STATUS	GPIO status

```

41 };
42
43 void gpio_event_string(char *buf, uint32_t events) {
44     for (uint i = 0; i < 4; i++) {
45         uint mask = (1 << i);
46         if (events & mask) {
47             // 将该事件字符串复制到用户缓冲区
48             const char *event_str = gpio_irq_str[i];
49             while (*event_str != '\0') {
50                 *buf++ = *event_str++;
51             }
52             events &= ~mask;
53
54             // 若有更多事件，则添加“，”
55             if (events) {
56                 *buf++ = ',';
57                 *buf++ = ' ';
58             }
59         }
60     }
61     *buf++ = '\0';
62 }

```

2.19.6. 寄存器列表

2.19.6.1. IO - 用户寄存器组

用户寄存器组的IO寄存器起始地址为 `0x40014000`（在SDK中定义为 `IO_BANK0_BASE`）。

表 283. `IO_BANK0` 寄存器列表

偏移量	名称	说明
0x000	<code>GPIO0_STATUS</code>	GPIO 状态
0x004	<code>GPIO0_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x008	<code>GPIO1_STATUS</code>	GPIO 状态
0x00c	<code>GPIO1_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x010	<code>GPIO2_STATUS</code>	GPIO 状态
0x014	<code>GPIO2_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x018	<code>GPIO3_STATUS</code>	GPIO 状态
0x01c	<code>GPIO3_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x020	<code>GPIO4_STATUS</code>	GPIO 状态
0x024	<code>GPIO4_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x028	<code>GPIO5_STATUS</code>	GPIO 状态
0x02c	<code>GPIO5_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x030	<code>GPIO6_STATUS</code>	GPIO 状态
0x034	<code>GPIO6_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x038	<code>GPIO7_STATUS</code>	GPIO 状态
0x03c	<code>GPIO7_CTRL</code>	GPIO 控制，包括功能选择和覆盖。
0x040	<code>GPIO8_STATUS</code>	GPIO 状态

Offset	Name	Info
0x044	GPIO8_CTRL	GPIO control including function select and overrides.
0x048	GPIO9_STATUS	GPIO status
0x04c	GPIO9_CTRL	GPIO control including function select and overrides.
0x050	GPIO10_STATUS	GPIO status
0x054	GPIO10_CTRL	GPIO control including function select and overrides.
0x058	GPIO11_STATUS	GPIO status
0x05c	GPIO11_CTRL	GPIO control including function select and overrides.
0x060	GPIO12_STATUS	GPIO status
0x064	GPIO12_CTRL	GPIO control including function select and overrides.
0x068	GPIO13_STATUS	GPIO status
0x06c	GPIO13_CTRL	GPIO control including function select and overrides.
0x070	GPIO14_STATUS	GPIO status
0x074	GPIO14_CTRL	GPIO control including function select and overrides.
0x078	GPIO15_STATUS	GPIO status
0x07c	GPIO15_CTRL	GPIO control including function select and overrides.
0x080	GPIO16_STATUS	GPIO status
0x084	GPIO16_CTRL	GPIO control including function select and overrides.
0x088	GPIO17_STATUS	GPIO status
0x08c	GPIO17_CTRL	GPIO control including function select and overrides.
0x090	GPIO18_STATUS	GPIO status
0x094	GPIO18_CTRL	GPIO control including function select and overrides.
0x098	GPIO19_STATUS	GPIO status
0x09c	GPIO19_CTRL	GPIO control including function select and overrides.
0x0a0	GPIO20_STATUS	GPIO status
0x0a4	GPIO20_CTRL	GPIO control including function select and overrides.
0x0a8	GPIO21_STATUS	GPIO status
0x0ac	GPIO21_CTRL	GPIO control including function select and overrides.
0x0b0	GPIO22_STATUS	GPIO status
0x0b4	GPIO22_CTRL	GPIO control including function select and overrides.
0x0b8	GPIO23_STATUS	GPIO status
0x0bc	GPIO23_CTRL	GPIO control including function select and overrides.
0x0c0	GPIO24_STATUS	GPIO status
0x0c4	GPIO24_CTRL	GPIO control including function select and overrides.
0x0c8	GPIO25_STATUS	GPIO status
0x0cc	GPIO25_CTRL	GPIO control including function select and overrides.
0x0d0	GPIO26_STATUS	GPIO status

偏移量	名称	说明
0x044	GPIO8_CTRL	GPIO 控制，包括功能选择和覆盖。
0x048	GPIO9_STATUS	GPIO 状态
0x04c	GPIO9_CTRL	GPIO 控制，包括功能选择和覆盖。
0x050	GPIO10_STATUS	GPIO 状态
0x054	GPIO10_CTRL	GPIO 控制，包括功能选择和覆盖。
0x058	GPIO11_STATUS	GPIO 状态
0x05c	GPIO11_CTRL	GPIO 控制，包括功能选择和覆盖。
0x060	GPIO12_STATUS	GPIO 状态
0x064	GPIO12_CTRL	GPIO 控制，包括功能选择和覆盖。
0x068	GPIO13_STATUS	GPIO 状态
0x06c	GPIO13_CTRL	GPIO 控制，包括功能选择和覆盖。
0x070	GPIO14_STATUS	GPIO 状态
0x074	GPIO14_CTRL	GPIO 控制，包括功能选择和覆盖。
0x078	GPIO15_STATUS	GPIO 状态
0x07c	GPIO15_CTRL	GPIO 控制，包括功能选择和覆盖。
0x080	GPIO16_STATUS	GPIO 状态
0x084	GPIO16_CTRL	GPIO 控制，包括功能选择和覆盖。
0x088	GPIO17_STATUS	GPIO 状态
0x08c	GPIO17_CTRL	GPIO 控制，包括功能选择和覆盖。
0x090	GPIO18_STATUS	GPIO 状态
0x094	GPIO18_CTRL	GPIO 控制，包括功能选择和覆盖。
0x098	GPIO19_STATUS	GPIO 状态
0x09c	GPIO19_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0a0	GPIO20_STATUS	GPIO 状态
0x0a4	GPIO20_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0a8	GPIO21_STATUS	GPIO 状态
0x0ac	GPIO21_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0b0	GPIO22_STATUS	GPIO 状态
0x0b4	GPIO22_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0b8	GPIO23_STATUS	GPIO 状态
0x0bc	GPIO23_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0c0	GPIO24_STATUS	GPIO 状态
0x0c4	GPIO24_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0c8	GPIO25_STATUS	GPIO 状态
0x0cc	GPIO25_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0d0	GPIO26_STATUS	GPIO 状态

Offset	Name	Info
0x0d4	GPIO26_CTRL	GPIO control including function select and overrides.
0x0d8	GPIO27_STATUS	GPIO status
0x0dc	GPIO27_CTRL	GPIO control including function select and overrides.
0x0e0	GPIO28_STATUS	GPIO status
0x0e4	GPIO28_CTRL	GPIO control including function select and overrides.
0x0e8	GPIO29_STATUS	GPIO status
0x0ec	GPIO29_CTRL	GPIO control including function select and overrides.
0x0f0	INTR0	Raw Interrupts
0x0f4	INTR1	Raw Interrupts
0x0f8	INTR2	Raw Interrupts
0x0fc	INTR3	Raw Interrupts
0x100	PROC0_INTE0	Interrupt Enable for proc0
0x104	PROC0_INTE1	Interrupt Enable for proc0
0x108	PROC0_INTE2	Interrupt Enable for proc0
0x10c	PROC0_INTE3	Interrupt Enable for proc0
0x110	PROC0_INTF0	Interrupt Force for proc0
0x114	PROC0_INTF1	Interrupt Force for proc0
0x118	PROC0_INTF2	Interrupt Force for proc0
0x11c	PROC0_INTF3	Interrupt Force for proc0
0x120	PROC0_INTS0	Interrupt status after masking & forcing for proc0
0x124	PROC0_INTS1	Interrupt status after masking & forcing for proc0
0x128	PROC0_INTS2	Interrupt status after masking & forcing for proc0
0x12c	PROC0_INTS3	Interrupt status after masking & forcing for proc0
0x130	PROC1_INTE0	Interrupt Enable for proc1
0x134	PROC1_INTE1	Interrupt Enable for proc1
0x138	PROC1_INTE2	Interrupt Enable for proc1
0x13c	PROC1_INTE3	Interrupt Enable for proc1
0x140	PROC1_INTF0	Interrupt Force for proc1
0x144	PROC1_INTF1	Interrupt Force for proc1
0x148	PROC1_INTF2	Interrupt Force for proc1
0x14c	PROC1_INTF3	Interrupt Force for proc1
0x150	PROC1_INTS0	Interrupt status after masking & forcing for proc1
0x154	PROC1_INTS1	Interrupt status after masking & forcing for proc1
0x158	PROC1_INTS2	Interrupt status after masking & forcing for proc1
0x15c	PROC1_INTS3	Interrupt status after masking & forcing for proc1
0x160	DORMANT_WAKE_INTE0	Interrupt Enable for dormant_wake

偏移量	名称	说明
0x0d4	GPIO26_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0d8	GPIO27_STATUS	GPIO 状态
0x0dc	GPIO27_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0e0	GPIO28_STATUS	GPIO 状态
0x0e4	GPIO28_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0e8	GPIO29_STATUS	GPIO 状态
0x0ec	GPIO29_CTRL	GPIO 控制，包括功能选择和覆盖。
0x0f0	中断 0	原始中断
0x0f4	中断 1	原始中断
0x0f8	中断 2	原始中断
0x0fc	中断 3	原始中断
0x100	PROC0_INTE0	proc0 的中断使能
0x104	PROC0_INTE1	proc0 的中断使能
0x108	PROC0_INTE2	proc0 的中断使能
0x10c	PROC0_INTE3	proc0 的中断使能
0x110	PROC0_INTFO	proc0 的中断强制
0x114	PROC0_INTF1	proc0 的中断强制
0x118	PROC0_INTF2	proc0 的中断强制
0x11c	PROC0_INTF3	proc0 的中断强制
0x120	PROC0_INTS0	proc0 的中断状态（掩码及强制后）
0x124	PROC0_INTS1	proc0 的中断状态（掩码及强制后）
0x128	PROC0_INTS2	proc0 的中断状态（掩码及强制后）
0x12c	PROC0_INTS3	proc0 的中断状态（掩码及强制后）
0x130	PROC1_INTE0	proc1 的中断使能
0x134	PROC1_INTE1	proc1 的中断使能
0x138	PROC1_INTE2	proc1 的中断使能
0x13c	PROC1_INTE3	proc1 的中断使能
0x140	PROC1_INTFO	proc1 的中断强制
0x144	PROC1_INTF1	proc1 的中断强制
0x148	PROC1_INTF2	proc1 的中断强制
0x14c	PROC1_INTF3	proc1 的中断强制
0x150	PROC1_INTS0	proc1 的中断状态（掩码及强制后）
0x154	PROC1_INTS1	proc1 的中断状态（掩码及强制后）
0x158	PROC1_INTS2	proc1 的中断状态（掩码及强制后）
0x15c	PROC1_INTS3	proc1 的中断状态（掩码及强制后）
0x160	DORMANT_WAKE_INTE0	dormant_wake 的中断使能

Offset	Name	Info
0x164	DORMANT_WAKE_INTE1	Interrupt Enable for dormant_wake
0x168	DORMANT_WAKE_INTE2	Interrupt Enable for dormant_wake
0x16c	DORMANT_WAKE_INTE3	Interrupt Enable for dormant_wake
0x170	DORMANT_WAKE_INTFO	Interrupt Force for dormant_wake
0x174	DORMANT_WAKE_INTF1	Interrupt Force for dormant_wake
0x178	DORMANT_WAKE_INTF2	Interrupt Force for dormant_wake
0x17c	DORMANT_WAKE_INTF3	Interrupt Force for dormant_wake
0x180	DORMANT_WAKE_INTSO	Interrupt status after masking & forcing for dormant_wake
0x184	DORMANT_WAKE_INTS1	Interrupt status after masking & forcing for dormant_wake
0x188	DORMANT_WAKE_INTS2	Interrupt status after masking & forcing for dormant_wake
0x18c	DORMANT_WAKE_INTS3	Interrupt status after masking & forcing for dormant_wake

IO_BANK0: GPIO0_STATUS, GPIO1_STATUS, ..., GPIO28_STATUS, GPIO29_STATUS Registers

Offsets: 0x000, 0x008, ..., 0x0e0, 0x0e8

Description

GPIO status

Table 284.
GPIO0_STATUS,
GPIO1_STATUS, ...,
GPIO28_STATUS,
GPIO29_STATUS
Registers

Bits	Description	Type	Reset
31:27	Reserved.	-	-
26	IRQTOPROC : interrupt to processors, after override is applied	RO	0x0
25	Reserved.	-	-
24	IRQFROMPAD : interrupt from pad before override is applied	RO	0x0
23:20	Reserved.	-	-
19	INTOPERI : input signal to peripheral, after override is applied	RO	0x0
18	Reserved.	-	-
17	INFROMPAD : input signal from pad, before override is applied	RO	0x0
16:14	Reserved.	-	-
13	OETOPAD : output enable to pad after register override is applied	RO	0x0
12	OEFROMPERI : output enable from selected peripheral, before register override is applied	RO	0x0
11:10	Reserved.	-	-
9	OUTTOPAD : output signal to pad after register override is applied	RO	0x0
8	OUTFROMPERI : output signal from selected peripheral, before register override is applied	RO	0x0
7:0	Reserved.	-	-

IO_BANK0: GPIO0_CTRL, GPIO1_CTRL, ..., GPIO28_CTRL, GPIO29_CTRL

偏移量	名称	说明
0x164	DORMANT_WAKE_INTE1	dormant_wake 的中断使能
0x168	DORMANT_WAKE_INTE2	dormant_wake 的中断使能
0x16c	DORMANT_WAKE_INTE3	dormant_wake 的中断使能
0x170	DORMANT_WAKE_INTFO	dormant_wake 的中断强制
0x174	DORMANT_WAKE_INTF1	dormant_wake 的中断强制
0x178	DORMANT_WAKE_INTF2	dormant_wake 的中断强制
0x17c	DORMANT_WAKE_INTF3	dormant_wake 的中断强制
0x180	DORMANT_WAKE_INTS0	dormant_wake 遮蔽及强制后的中断状态
0x184	DORMANT_WAKE_INTS1	dormant_wake 遮蔽及强制后的中断状态
0x188	DORMANT_WAKE_INTS2	dormant_wake 遮蔽及强制后的中断状态
0x18c	DORMANT_WAKE_INTS3	dormant_wake 遮蔽及强制后的中断状态

IO_BANK0: GPIO0_STATUS, GPIO1_STATUS, ..., GPIO28_STATUS, GPIO29_STATUS 寄存器

偏移量: 0x000, 0x008, ..., 0x0e0, 0x0e8

描述

GPIO 状态

表 284.
GPIO0_STATUS,
GPIO1_STATUS, ...,
GPIO28_STATUS,
GPIO29_STATUS
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	IRQTOPROC : 应用覆盖后传递给处理器的中断	只读	0x0
25	保留。	-	-
24	IRQFROMPAD : 应用覆盖前来自引脚的中断	只读	0x0
23:20	保留。	-	-
19	INTOPERI : 应用覆盖后传入外设的信号	只读	0x0
18	保留。	-	-
17	INFROMPAD : 应用覆盖前来自引脚的输入信号	只读	0x0
16:14	保留。	-	-
13	OETOPAD : 应用寄存器覆盖后传给引脚的输出使能	只读	0x0
12	OEFROMPERI : 应用寄存器覆盖前来自选定外设的输出使能	只读	0x0
11:10	保留。	-	-
9	OUTTOPAD : 应用寄存器覆盖后传给引脚的输出信号	只读	0x0
8	OUTFROMPERI : 所选外设的输出信号，在寄存器覆盖生效前	只读	0x0
7:0	保留。	-	-

IO_BANK0: GPIO0_CTRL、GPIO1_CTRL、...、GPIO28_CTRL、GPIO29_CTRL

Registers

Offsets: 0x004, 0x00c, ..., 0x0e4, 0x0ec

Description

GPIO control including function select and overrides.

*Table 285.
GPIO0_CTRL,
GPIO1_CTRL, ...,
GPIO28_CTRL,
GPIO29_CTRL
Registers*

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:28	IRQOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: don't invert the interrupt		
	0x1 → INVERT: invert the interrupt		
	0x2 → LOW: drive interrupt low		
	0x3 → HIGH: drive interrupt high		
27:18	Reserved.	-	-
17:16	INOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: don't invert the peri input		
	0x1 → INVERT: invert the peri input		
	0x2 → LOW: drive peri input low		
	0x3 → HIGH: drive peri input high		
15:14	Reserved.	-	-
13:12	OEOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: drive output enable from peripheral signal selected by funcsel		
	0x1 → INVERT: drive output enable from inverse of peripheral signal selected by funcsel		
	0x2 → DISABLE: disable output		
	0x3 → ENABLE: enable output		
11:10	Reserved.	-	-
9:8	OUTOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: drive output from peripheral signal selected by funcsel		
	0x1 → INVERT: drive output from inverse of peripheral signal selected by funcsel		
	0x2 → LOW: drive output low		
	0x3 → HIGH: drive output high		
7:5	Reserved.	-	-

寄存器

偏移量：0x004、0x00c、...、0x0e4、0x0ec

描述

GPIO 控制，包括功能选择和覆盖。

表285。
GPIO0_CTRL,
GPIO1_CTRL, ...
GPIO28_CTRL,
GPIO29_CTRL
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:28	IRQOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：中断不反转		
	0x1 → INVERT：中断反转		
	0x2 → LOW：中断拉低		
	0x3 → HIGH：中断拉高		
27:18	保留。	-	-
17:16	INOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：外设输入不反转		
	0x1 → INVERT：外设输入反转		
	0x2 → LOW：外设输入拉低		
	0x3 → HIGH：外设输入拉高		
15:14	保留。	-	-
13:12	OEOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出使能，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出使能，由funcsel选择的外设信号的反相信号控制		
	0x2 → DISABLE：禁用输出		
	0x3 → ENABLE：使能输出		
11:10	保留。	-	-
9:8	OUTOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出，由funcsel选择的外设信号的反相信号控制		
	0x2 → LOW：驱动输出低电平		
	0x3 → HIGH：驱动输出高电平		
7:5	保留。	-	-

Bits	Description	Type	Reset
4:0	FUNCSEL: Function select. 31 == NULL. See GPIO function table for available functions.	RW	0x1f

IO_BANK0: INTR0 Register

Offset: 0x0f0

Description

Raw Interrupts

Table 286. INTR0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	WC	0x0
30	GPIO7_EDGE_LOW	WC	0x0
29	GPIO7_LEVEL_HIGH	RO	0x0
28	GPIO7_LEVEL_LOW	RO	0x0
27	GPIO6_EDGE_HIGH	WC	0x0
26	GPIO6_EDGE_LOW	WC	0x0
25	GPIO6_LEVEL_HIGH	RO	0x0
24	GPIO6_LEVEL_LOW	RO	0x0
23	GPIO5_EDGE_HIGH	WC	0x0
22	GPIO5_EDGE_LOW	WC	0x0
21	GPIO5_LEVEL_HIGH	RO	0x0
20	GPIO5_LEVEL_LOW	RO	0x0
19	GPIO4_EDGE_HIGH	WC	0x0
18	GPIO4_EDGE_LOW	WC	0x0
17	GPIO4_LEVEL_HIGH	RO	0x0
16	GPIO4_LEVEL_LOW	RO	0x0
15	GPIO3_EDGE_HIGH	WC	0x0
14	GPIO3_EDGE_LOW	WC	0x0
13	GPIO3_LEVEL_HIGH	RO	0x0
12	GPIO3_LEVEL_LOW	RO	0x0
11	GPIO2_EDGE_HIGH	WC	0x0
10	GPIO2_EDGE_LOW	WC	0x0
9	GPIO2_LEVEL_HIGH	RO	0x0
8	GPIO2_LEVEL_LOW	RO	0x0
7	GPIO1_EDGE_HIGH	WC	0x0
6	GPIO1_EDGE_LOW	WC	0x0
5	GPIO1_LEVEL_HIGH	RO	0x0
4	GPIO1_LEVEL_LOW	RO	0x0
3	GPIO0_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
4:0	FUNCSEL : 功能选择。31表示NULL。具体可用功能请参见GPIO功能表。	读写	0x1f

IO_BANK0: INTRO寄存器

偏移: 0x0f0

描述

原始中断

表 286. INTRO
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	WC	0x0
30	GPIO7_EDGE_LOW	WC	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	WC	0x0
26	GPIO6_EDGE_LOW	WC	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	WC	0x0
22	GPIO5_EDGE_LOW	WC	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	WC	0x0
18	GPIO4_EDGE_LOW	WC	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	WC	0x0
14	GPIO3_EDGE_LOW	WC	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	WC	0x0
10	GPIO2_EDGE_LOW	WC	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	WC	0x0
6	GPIO1_EDGE_LOW	WC	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	WC	0x0

Bits	Description	Type	Reset
2	GPIO0_EDGE_LOW	WC	0x0
1	GPIO0_LEVEL_HIGH	RO	0x0
0	GPIO0_LEVEL_LOW	RO	0x0

IO_BANK0: INTR1 Register

Offset: 0x0f4

Description

Raw Interrupts

Table 287. INTR1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	WC	0x0
30	GPIO15_EDGE_LOW	WC	0x0
29	GPIO15_LEVEL_HIGH	RO	0x0
28	GPIO15_LEVEL_LOW	RO	0x0
27	GPIO14_EDGE_HIGH	WC	0x0
26	GPIO14_EDGE_LOW	WC	0x0
25	GPIO14_LEVEL_HIGH	RO	0x0
24	GPIO14_LEVEL_LOW	RO	0x0
23	GPIO13_EDGE_HIGH	WC	0x0
22	GPIO13_EDGE_LOW	WC	0x0
21	GPIO13_LEVEL_HIGH	RO	0x0
20	GPIO13_LEVEL_LOW	RO	0x0
19	GPIO12_EDGE_HIGH	WC	0x0
18	GPIO12_EDGE_LOW	WC	0x0
17	GPIO12_LEVEL_HIGH	RO	0x0
16	GPIO12_LEVEL_LOW	RO	0x0
15	GPIO11_EDGE_HIGH	WC	0x0
14	GPIO11_EDGE_LOW	WC	0x0
13	GPIO11_LEVEL_HIGH	RO	0x0
12	GPIO11_LEVEL_LOW	RO	0x0
11	GPIO10_EDGE_HIGH	WC	0x0
10	GPIO10_EDGE_LOW	WC	0x0
9	GPIO10_LEVEL_HIGH	RO	0x0
8	GPIO10_LEVEL_LOW	RO	0x0
7	GPIO9_EDGE_HIGH	WC	0x0
6	GPIO9_EDGE_LOW	WC	0x0
5	GPIO9_LEVEL_HIGH	RO	0x0

位	描述	类型	复位值
2	GPIO0_EDGE_LOW	WC	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0：INTR1寄存器

偏移量：0x0f4

描述

原始中断

表 287. INTR1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	WC	0x0
30	GPIO15_EDGE_LOW	WC	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	WC	0x0
26	GPIO14_EDGE_LOW	WC	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	WC	0x0
22	GPIO13_EDGE_LOW	WC	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	WC	0x0
18	GPIO12_EDGE_LOW	WC	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	WC	0x0
14	GPIO11_EDGE_LOW	WC	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	WC	0x0
10	GPIO10_EDGE_LOW	WC	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	WC	0x0
6	GPIO9_EDGE_LOW	WC	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0

Bits	Description	Type	Reset
4	GPIO9_LEVEL_LOW	RO	0x0
3	GPIO8_EDGE_HIGH	WC	0x0
2	GPIO8_EDGE_LOW	WC	0x0
1	GPIO8_LEVEL_HIGH	RO	0x0
0	GPIO8_LEVEL_LOW	RO	0x0

IO_BANK0: INTR2 Register

Offset: 0x0f8

Description

Raw Interrupts

Table 288. INTR2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	WC	0x0
30	GPIO23_EDGE_LOW	WC	0x0
29	GPIO23_LEVEL_HIGH	RO	0x0
28	GPIO23_LEVEL_LOW	RO	0x0
27	GPIO22_EDGE_HIGH	WC	0x0
26	GPIO22_EDGE_LOW	WC	0x0
25	GPIO22_LEVEL_HIGH	RO	0x0
24	GPIO22_LEVEL_LOW	RO	0x0
23	GPIO21_EDGE_HIGH	WC	0x0
22	GPIO21_EDGE_LOW	WC	0x0
21	GPIO21_LEVEL_HIGH	RO	0x0
20	GPIO21_LEVEL_LOW	RO	0x0
19	GPIO20_EDGE_HIGH	WC	0x0
18	GPIO20_EDGE_LOW	WC	0x0
17	GPIO20_LEVEL_HIGH	RO	0x0
16	GPIO20_LEVEL_LOW	RO	0x0
15	GPIO19_EDGE_HIGH	WC	0x0
14	GPIO19_EDGE_LOW	WC	0x0
13	GPIO19_LEVEL_HIGH	RO	0x0
12	GPIO19_LEVEL_LOW	RO	0x0
11	GPIO18_EDGE_HIGH	WC	0x0
10	GPIO18_EDGE_LOW	WC	0x0
9	GPIO18_LEVEL_HIGH	RO	0x0
8	GPIO18_LEVEL_LOW	RO	0x0
7	GPIO17_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	WC	0x0
2	GPIO8_EDGE_LOW	WC	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: INTR2 寄存器

偏移: 0x0f8

描述

原始中断

表 288. INTR2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	WC	0x0
30	GPIO23_EDGE_LOW	WC	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	WC	0x0
26	GPIO22_EDGE_LOW	WC	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	WC	0x0
22	GPIO21_EDGE_LOW	WC	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	WC	0x0
18	GPIO20_EDGE_LOW	WC	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	WC	0x0
14	GPIO19_EDGE_LOW	WC	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	WC	0x0
10	GPIO18_EDGE_LOW	WC	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	WC	0x0

Bits	Description	Type	Reset
6	GPIO17_EDGE_LOW	WC	0x0
5	GPIO17_LEVEL_HIGH	RO	0x0
4	GPIO17_LEVEL_LOW	RO	0x0
3	GPIO16_EDGE_HIGH	WC	0x0
2	GPIO16_EDGE_LOW	WC	0x0
1	GPIO16_LEVEL_HIGH	RO	0x0
0	GPIO16_LEVEL_LOW	RO	0x0

IO_BANK0: INTR3 Register

Offset: 0x0fc

Description

Raw Interrupts

Table 289. INTR3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	WC	0x0
22	GPIO29_EDGE_LOW	WC	0x0
21	GPIO29_LEVEL_HIGH	RO	0x0
20	GPIO29_LEVEL_LOW	RO	0x0
19	GPIO28_EDGE_HIGH	WC	0x0
18	GPIO28_EDGE_LOW	WC	0x0
17	GPIO28_LEVEL_HIGH	RO	0x0
16	GPIO28_LEVEL_LOW	RO	0x0
15	GPIO27_EDGE_HIGH	WC	0x0
14	GPIO27_EDGE_LOW	WC	0x0
13	GPIO27_LEVEL_HIGH	RO	0x0
12	GPIO27_LEVEL_LOW	RO	0x0
11	GPIO26_EDGE_HIGH	WC	0x0
10	GPIO26_EDGE_LOW	WC	0x0
9	GPIO26_LEVEL_HIGH	RO	0x0
8	GPIO26_LEVEL_LOW	RO	0x0
7	GPIO25_EDGE_HIGH	WC	0x0
6	GPIO25_EDGE_LOW	WC	0x0
5	GPIO25_LEVEL_HIGH	RO	0x0
4	GPIO25_LEVEL_LOW	RO	0x0
3	GPIO24_EDGE_HIGH	WC	0x0
2	GPIO24_EDGE_LOW	WC	0x0

位	描述	类型	复位值
6	GPIO17_EDGE_LOW	WC	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	WC	0x0
2	GPIO16_EDGE_LOW	WC	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: INTR3 寄存器

偏移: 0x0fc

描述

原始中断

表 289. INTR3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	WC	0x0
22	GPIO29_EDGE_LOW	WC	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	WC	0x0
18	GPIO28_EDGE_LOW	WC	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	WC	0x0
14	GPIO27_EDGE_LOW	WC	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	WC	0x0
10	GPIO26_EDGE_LOW	WC	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	WC	0x0
6	GPIO25_EDGE_LOW	WC	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	WC	0x0
2	GPIO24_EDGE_LOW	WC	0x0

Bits	Description	Type	Reset
1	GPIO24_LEVEL_HIGH	RO	0x0
0	GPIO24_LEVEL_LOW	RO	0x0

IO_BANK0: PROCO_INTE0 Register

Offset: 0x100

Description

Interrupt Enable for proc0

Table 290.
PROCO_INTE0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0
10	GPIO2_EDGE_LOW	RW	0x0
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0
7	GPIO1_EDGE_HIGH	RW	0x0
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0
4	GPIO1_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTE0 寄存器

偏移: 0x100

描述

proc0 的中断使能

表 290.
PROC0_INTE0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTE1 Register

Offset: 0x104

Description

Interrupt Enable for proc0

Table 291.
PROC0_INTE1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0
12	GPIO11_LEVEL_LOW	RW	0x0
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0
9	GPIO10_LEVEL_HIGH	RW	0x0
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0
6	GPIO9_EDGE_LOW	RW	0x0

位	描述	类型	复位值
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTE1 寄存器

偏移: 0x104

描述

proc0 的中断使能

表 291.
PROC0_INTE1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTE2 Register

Offset: 0x108

Description

Interrupt Enable for proc0

Table 292.
PROC0_INTE2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0
14	GPIO19_EDGE_LOW	RW	0x0
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0
11	GPIO18_EDGE_HIGH	RW	0x0
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0
8	GPIO18_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0：PROC0_INTE2 寄存器

偏移: 0x108

描述

proc0 的中断使能

表 292.
PROC0_INTE2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTE3 Register

Offset: 0x10c

Description

Interrupt Enable for proc0

Table 293.
PROC0_INTE3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0
9	GPIO26_LEVEL_HIGH	RW	0x0
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0
6	GPIO25_EDGE_LOW	RW	0x0
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0
3	GPIO24_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0：PROC0_INTE3 寄存器

偏移: 0x10c

描述

proc0 的中断使能

表293。
PROC0_INTE3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTF0 Register

Offset: 0x110

Description

Interrupt Force for proc0

Table 294.
PROC0_INTF0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0
10	GPIO2_EDGE_LOW	RW	0x0
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0
7	GPIO1_EDGE_HIGH	RW	0x0
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTFO 寄存器

偏移量: 0x110

描述

proc0 的中断强制

表294。
PROC0_INTFO 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
4	GPIO1_LEVEL_LOW	RW	0x0
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: PROCO_INTF1 Register

Offset: 0x114

Description

Interrupt Force for proc0

Table 295.
PROCO_INTF1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0
12	GPIO11_LEVEL_LOW	RW	0x0
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0
9	GPIO10_LEVEL_HIGH	RW	0x0
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF1 寄存器

偏移量: 0x114

描述

proc0 的中断强制

表295。
PROC0_INTF1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
6	GPIO9_EDGE_LOW	RW	0x0
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTF2 Register

Offset: 0x118

Description

Interrupt Force for proc0

Table 296.
PROC0_INTF2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0
14	GPIO19_EDGE_LOW	RW	0x0
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0
11	GPIO18_EDGE_HIGH	RW	0x0
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF2 寄存器

偏移量: 0x118

描述

proc0 的中断强制

表296。
PROC0_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
8	GPIO18_LEVEL_LOW	RW	0x0
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: PROC0_INTF3 Register

Offset: 0x11c

Description

Interrupt Force for proc0

Table 297.
PROC0_INTF3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0
9	GPIO26_LEVEL_HIGH	RW	0x0
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0
6	GPIO25_EDGE_LOW	RW	0x0
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTF3 寄存器

偏移量: 0x11c

描述

proc0 的中断强制

表297。
PROC0_INTF3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
3	GPIO24_EDGE_HIGH	RW	0x0
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: PROCO_INTS0 Register

Offset: 0x120

Description

Interrupt status after masking & forcing for proc0

Table 298.
PROCO_INTS0
Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RO	0x0
30	GPIO7_EDGE_LOW	RO	0x0
29	GPIO7_LEVEL_HIGH	RO	0x0
28	GPIO7_LEVEL_LOW	RO	0x0
27	GPIO6_EDGE_HIGH	RO	0x0
26	GPIO6_EDGE_LOW	RO	0x0
25	GPIO6_LEVEL_HIGH	RO	0x0
24	GPIO6_LEVEL_LOW	RO	0x0
23	GPIO5_EDGE_HIGH	RO	0x0
22	GPIO5_EDGE_LOW	RO	0x0
21	GPIO5_LEVEL_HIGH	RO	0x0
20	GPIO5_LEVEL_LOW	RO	0x0
19	GPIO4_EDGE_HIGH	RO	0x0
18	GPIO4_EDGE_LOW	RO	0x0
17	GPIO4_LEVEL_HIGH	RO	0x0
16	GPIO4_LEVEL_LOW	RO	0x0
15	GPIO3_EDGE_HIGH	RO	0x0
14	GPIO3_EDGE_LOW	RO	0x0
13	GPIO3_LEVEL_HIGH	RO	0x0
12	GPIO3_LEVEL_LOW	RO	0x0
11	GPIO2_EDGE_HIGH	RO	0x0
10	GPIO2_EDGE_LOW	RO	0x0
9	GPIO2_LEVEL_HIGH	RO	0x0
8	GPIO2_LEVEL_LOW	RO	0x0
7	GPIO1_EDGE_HIGH	RO	0x0
6	GPIO1_EDGE_LOW	RO	0x0

位	描述	类型	复位值
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC0_INTS0 寄存器

偏移: 0x120

说明

proc0 的中断状态（掩码及强制后）

表 298
PROC0_INTS0
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0

Bits	Description	Type	Reset
5	GPIO1_LEVEL_HIGH	RO	0x0
4	GPIO1_LEVEL_LOW	RO	0x0
3	GPIO0_EDGE_HIGH	RO	0x0
2	GPIO0_EDGE_LOW	RO	0x0
1	GPIO0_LEVEL_HIGH	RO	0x0
0	GPIO0_LEVEL_LOW	RO	0x0

IO_BANK0: PROC0_INTS1 Register

Offset: 0x124

Description

Interrupt status after masking & forcing for proc0

Table 299.
PROC0_INTS1
Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RO	0x0
30	GPIO15_EDGE_LOW	RO	0x0
29	GPIO15_LEVEL_HIGH	RO	0x0
28	GPIO15_LEVEL_LOW	RO	0x0
27	GPIO14_EDGE_HIGH	RO	0x0
26	GPIO14_EDGE_LOW	RO	0x0
25	GPIO14_LEVEL_HIGH	RO	0x0
24	GPIO14_LEVEL_LOW	RO	0x0
23	GPIO13_EDGE_HIGH	RO	0x0
22	GPIO13_EDGE_LOW	RO	0x0
21	GPIO13_LEVEL_HIGH	RO	0x0
20	GPIO13_LEVEL_LOW	RO	0x0
19	GPIO12_EDGE_HIGH	RO	0x0
18	GPIO12_EDGE_LOW	RO	0x0
17	GPIO12_LEVEL_HIGH	RO	0x0
16	GPIO12_LEVEL_LOW	RO	0x0
15	GPIO11_EDGE_HIGH	RO	0x0
14	GPIO11_EDGE_LOW	RO	0x0
13	GPIO11_LEVEL_HIGH	RO	0x0
12	GPIO11_LEVEL_LOW	RO	0x0
11	GPIO10_EDGE_HIGH	RO	0x0
10	GPIO10_EDGE_LOW	RO	0x0
9	GPIO10_LEVEL_HIGH	RO	0x0
8	GPIO10_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTS1 寄存器

偏移: 0x124

说明

proc0 的中断状态 (掩码及强制后)

表 299
PROC0_INTS1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
7	GPIO9_EDGE_HIGH	RO	0x0
6	GPIO9_EDGE_LOW	RO	0x0
5	GPIO9_LEVEL_HIGH	RO	0x0
4	GPIO9_LEVEL_LOW	RO	0x0
3	GPIO8_EDGE_HIGH	RO	0x0
2	GPIO8_EDGE_LOW	RO	0x0
1	GPIO8_LEVEL_HIGH	RO	0x0
0	GPIO8_LEVEL_LOW	RO	0x0

IO_BANK0: PROC0_INTS2 Register

Offset: 0x128

Description

Interrupt status after masking & forcing for proc0

Table 300.
PROC0_INTS2
Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RO	0x0
30	GPIO23_EDGE_LOW	RO	0x0
29	GPIO23_LEVEL_HIGH	RO	0x0
28	GPIO23_LEVEL_LOW	RO	0x0
27	GPIO22_EDGE_HIGH	RO	0x0
26	GPIO22_EDGE_LOW	RO	0x0
25	GPIO22_LEVEL_HIGH	RO	0x0
24	GPIO22_LEVEL_LOW	RO	0x0
23	GPIO21_EDGE_HIGH	RO	0x0
22	GPIO21_EDGE_LOW	RO	0x0
21	GPIO21_LEVEL_HIGH	RO	0x0
20	GPIO21_LEVEL_LOW	RO	0x0
19	GPIO20_EDGE_HIGH	RO	0x0
18	GPIO20_EDGE_LOW	RO	0x0
17	GPIO20_LEVEL_HIGH	RO	0x0
16	GPIO20_LEVEL_LOW	RO	0x0
15	GPIO19_EDGE_HIGH	RO	0x0
14	GPIO19_EDGE_LOW	RO	0x0
13	GPIO19_LEVEL_HIGH	RO	0x0
12	GPIO19_LEVEL_LOW	RO	0x0
11	GPIO18_EDGE_HIGH	RO	0x0
10	GPIO18_EDGE_LOW	RO	0x0

位	描述	类型	复位值
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0：PROC0_INTS2 寄存器

偏移: 0x128

说明

proc0 的中断状态（掩码及强制后）

表 300
PROC0_INTS2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0

Bits	Description	Type	Reset
9	GPIO18_LEVEL_HIGH	RO	0x0
8	GPIO18_LEVEL_LOW	RO	0x0
7	GPIO17_EDGE_HIGH	RO	0x0
6	GPIO17_EDGE_LOW	RO	0x0
5	GPIO17_LEVEL_HIGH	RO	0x0
4	GPIO17_LEVEL_LOW	RO	0x0
3	GPIO16_EDGE_HIGH	RO	0x0
2	GPIO16_EDGE_LOW	RO	0x0
1	GPIO16_LEVEL_HIGH	RO	0x0
0	GPIO16_LEVEL_LOW	RO	0x0

IO_BANK0: PROC0_INTS3 Register

Offset: 0x12c

Description

Interrupt status after masking & forcing for proc0

Table 301.
PROC0_INTS3
Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RO	0x0
22	GPIO29_EDGE_LOW	RO	0x0
21	GPIO29_LEVEL_HIGH	RO	0x0
20	GPIO29_LEVEL_LOW	RO	0x0
19	GPIO28_EDGE_HIGH	RO	0x0
18	GPIO28_EDGE_LOW	RO	0x0
17	GPIO28_LEVEL_HIGH	RO	0x0
16	GPIO28_LEVEL_LOW	RO	0x0
15	GPIO27_EDGE_HIGH	RO	0x0
14	GPIO27_EDGE_LOW	RO	0x0
13	GPIO27_LEVEL_HIGH	RO	0x0
12	GPIO27_LEVEL_LOW	RO	0x0
11	GPIO26_EDGE_HIGH	RO	0x0
10	GPIO26_EDGE_LOW	RO	0x0
9	GPIO26_LEVEL_HIGH	RO	0x0
8	GPIO26_LEVEL_LOW	RO	0x0
7	GPIO25_EDGE_HIGH	RO	0x0
6	GPIO25_EDGE_LOW	RO	0x0
5	GPIO25_LEVEL_HIGH	RO	0x0

位	描述	类型	复位值
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: PROC0_INTS3 寄存器

偏移: 0x12c

说明

proc0 的中断状态 (掩码及强制后)

表 301
PROC0_INTS3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0

Bits	Description	Type	Reset
4	GPIO25_LEVEL_LOW	RO	0x0
3	GPIO24_EDGE_HIGH	RO	0x0
2	GPIO24_EDGE_LOW	RO	0x0
1	GPIO24_LEVEL_HIGH	RO	0x0
0	GPIO24_LEVEL_LOW	RO	0x0

IO_BANK0: PROC1_INTE0 Register

Offset: 0x130

Description

Interrupt Enable for proc1

Table 302.
PROC1_INTE0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0
10	GPIO2_EDGE_LOW	RW	0x0
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0
7	GPIO1_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTE0 寄存器

偏移: 0x130

说明

proc1 的中断使能

表 302。
PROC1_INTE0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0
4	GPIO1_LEVEL_LOW	RW	0x0
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTE1 Register

Offset: 0x134

Description

Interrupt Enable for proc1

Table 303.
PROC1_INTE1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0
12	GPIO11_LEVEL_LOW	RW	0x0
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0
9	GPIO10_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE1 寄存器

偏移量: 0x134

说明

proc1 的中断使能

表 303。
PROC1_INTE1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0
6	GPIO9_EDGE_LOW	RW	0x0
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTE2 Register

Offset: 0x138

Description

Interrupt Enable for proc1

Table 304.
PROC1_INTE2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0
14	GPIO19_EDGE_LOW	RW	0x0
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0
11	GPIO18_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE2 寄存器

偏移量: 0x138

说明

proc1 的中断使能

表 304。
PROC1_INTE2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0
8	GPIO18_LEVEL_LOW	RW	0x0
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTE3 Register

Offset: 0x13c

Description

Interrupt Enable for proc1

Table 305.
PROC1_INTE3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0
9	GPIO26_LEVEL_HIGH	RW	0x0
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0
6	GPIO25_EDGE_LOW	RW	0x0

位	描述	类型	复位值
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTE3 寄存器

偏移量: 0x13c

说明

proc1 的中断使能

表 305。
PROC1_INTE3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0
3	GPIO24_EDGE_HIGH	RW	0x0
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTF0 Register

Offset: 0x140

Description

Interrupt Force for proc1

Table 306.
PROC1_INTF0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0
10	GPIO2_EDGE_LOW	RW	0x0
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTO 寄存器

偏移量: 0x140

描述

proc1 的中断强制

表 306。
PROC1_INTO 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
7	GPIO1_EDGE_HIGH	RW	0x0
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0
4	GPIO1_LEVEL_LOW	RW	0x0
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTF1 Register

Offset: 0x144

Description

Interrupt Force for proc1

Table 307.
PROC1_INTF1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0
12	GPIO11_LEVEL_LOW	RW	0x0
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0

位	描述	类型	复位值
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0：PROC1_INTF1 寄存器

偏移：0x144

描述

proc1 的中断强制

表 307
PROC1_INTF1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
9	GPIO10_LEVEL_HIGH	RW	0x0
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0
6	GPIO9_EDGE_LOW	RW	0x0
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTF2 Register

Offset: 0x148

Description

Interrupt Force for proc1

Table 308.
PROC1_INTF2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0
14	GPIO19_EDGE_LOW	RW	0x0
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTF2 寄存器

偏移: 0x148

描述

proc1 的中断强制

表 308
PROC1_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
11	GPIO18_EDGE_HIGH	RW	0x0
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0
8	GPIO18_LEVEL_LOW	RW	0x0
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTF3 Register

Offset: 0x14c

Description

Interrupt Force for proc1

Table 309.
PROC1_INTF3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0
9	GPIO26_LEVEL_HIGH	RW	0x0
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0：PROC1_INTF3 寄存器

偏移：0x14c

描述

proc1 的中断强制

表 309
PROC1_INTF3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
6	GPIO25_EDGE_LOW	RW	0x0
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0
3	GPIO24_EDGE_HIGH	RW	0x0
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: PROC1_INTS0 Register

Offset: 0x150

Description

Interrupt status after masking & forcing for proc1

Table 310.
PROC1_INTS0
Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RO	0x0
30	GPIO7_EDGE_LOW	RO	0x0
29	GPIO7_LEVEL_HIGH	RO	0x0
28	GPIO7_LEVEL_LOW	RO	0x0
27	GPIO6_EDGE_HIGH	RO	0x0
26	GPIO6_EDGE_LOW	RO	0x0
25	GPIO6_LEVEL_HIGH	RO	0x0
24	GPIO6_LEVEL_LOW	RO	0x0
23	GPIO5_EDGE_HIGH	RO	0x0
22	GPIO5_EDGE_LOW	RO	0x0
21	GPIO5_LEVEL_HIGH	RO	0x0
20	GPIO5_LEVEL_LOW	RO	0x0
19	GPIO4_EDGE_HIGH	RO	0x0
18	GPIO4_EDGE_LOW	RO	0x0
17	GPIO4_LEVEL_HIGH	RO	0x0
16	GPIO4_LEVEL_LOW	RO	0x0
15	GPIO3_EDGE_HIGH	RO	0x0
14	GPIO3_EDGE_LOW	RO	0x0
13	GPIO3_LEVEL_HIGH	RO	0x0
12	GPIO3_LEVEL_LOW	RO	0x0
11	GPIO2_EDGE_HIGH	RO	0x0
10	GPIO2_EDGE_LOW	RO	0x0
9	GPIO2_LEVEL_HIGH	RO	0x0

位	描述	类型	复位值
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: PROC1_INTS0 寄存器

偏移: 0x150

描述

proc1 的中断状态（掩码及强制后）

表 310
PROC1_INTS0
寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0

Bits	Description	Type	Reset
8	GPIO2_LEVEL_LOW	RO	0x0
7	GPIO1_EDGE_HIGH	RO	0x0
6	GPIO1_EDGE_LOW	RO	0x0
5	GPIO1_LEVEL_HIGH	RO	0x0
4	GPIO1_LEVEL_LOW	RO	0x0
3	GPIO0_EDGE_HIGH	RO	0x0
2	GPIO0_EDGE_LOW	RO	0x0
1	GPIO0_LEVEL_HIGH	RO	0x0
0	GPIO0_LEVEL_LOW	RO	0x0

IO_BANK0: PROC1_INTS1 Register

Offset: 0x154

Description

Interrupt status after masking & forcing for proc1

Table 311.
PROC1_INTS1
Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RO	0x0
30	GPIO15_EDGE_LOW	RO	0x0
29	GPIO15_LEVEL_HIGH	RO	0x0
28	GPIO15_LEVEL_LOW	RO	0x0
27	GPIO14_EDGE_HIGH	RO	0x0
26	GPIO14_EDGE_LOW	RO	0x0
25	GPIO14_LEVEL_HIGH	RO	0x0
24	GPIO14_LEVEL_LOW	RO	0x0
23	GPIO13_EDGE_HIGH	RO	0x0
22	GPIO13_EDGE_LOW	RO	0x0
21	GPIO13_LEVEL_HIGH	RO	0x0
20	GPIO13_LEVEL_LOW	RO	0x0
19	GPIO12_EDGE_HIGH	RO	0x0
18	GPIO12_EDGE_LOW	RO	0x0
17	GPIO12_LEVEL_HIGH	RO	0x0
16	GPIO12_LEVEL_LOW	RO	0x0
15	GPIO11_EDGE_HIGH	RO	0x0
14	GPIO11_EDGE_LOW	RO	0x0
13	GPIO11_LEVEL_HIGH	RO	0x0
12	GPIO11_LEVEL_LOW	RO	0x0
11	GPIO10_EDGE_HIGH	RO	0x0

位	描述	类型	复位值
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS1 寄存器

偏移: 0x154

描述

proc1 的中断状态（掩码及强制后）

表311
PROC1_INTS1
寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0

Bits	Description	Type	Reset
10	GPIO10_EDGE_LOW	RO	0x0
9	GPIO10_LEVEL_HIGH	RO	0x0
8	GPIO10_LEVEL_LOW	RO	0x0
7	GPIO9_EDGE_HIGH	RO	0x0
6	GPIO9_EDGE_LOW	RO	0x0
5	GPIO9_LEVEL_HIGH	RO	0x0
4	GPIO9_LEVEL_LOW	RO	0x0
3	GPIO8_EDGE_HIGH	RO	0x0
2	GPIO8_EDGE_LOW	RO	0x0
1	GPIO8_LEVEL_HIGH	RO	0x0
0	GPIO8_LEVEL_LOW	RO	0x0

IO_BANK0: PROC1_INTS2 Register

Offset: 0x158

Description

Interrupt status after masking & forcing for proc1

Table 312.
PROC1_INTS2
Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RO	0x0
30	GPIO23_EDGE_LOW	RO	0x0
29	GPIO23_LEVEL_HIGH	RO	0x0
28	GPIO23_LEVEL_LOW	RO	0x0
27	GPIO22_EDGE_HIGH	RO	0x0
26	GPIO22_EDGE_LOW	RO	0x0
25	GPIO22_LEVEL_HIGH	RO	0x0
24	GPIO22_LEVEL_LOW	RO	0x0
23	GPIO21_EDGE_HIGH	RO	0x0
22	GPIO21_EDGE_LOW	RO	0x0
21	GPIO21_LEVEL_HIGH	RO	0x0
20	GPIO21_LEVEL_LOW	RO	0x0
19	GPIO20_EDGE_HIGH	RO	0x0
18	GPIO20_EDGE_LOW	RO	0x0
17	GPIO20_LEVEL_HIGH	RO	0x0
16	GPIO20_LEVEL_LOW	RO	0x0
15	GPIO19_EDGE_HIGH	RO	0x0
14	GPIO19_EDGE_LOW	RO	0x0
13	GPIO19_LEVEL_HIGH	RO	0x0

位	描述	类型	复位值
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS2寄存器

偏移量: 0x158

说明

proc1 的中断状态 (掩码及强制后)

表312
PROC1_INTS2
寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0

Bits	Description	Type	Reset
12	GPIO19_LEVEL_LOW	RO	0x0
11	GPIO18_EDGE_HIGH	RO	0x0
10	GPIO18_EDGE_LOW	RO	0x0
9	GPIO18_LEVEL_HIGH	RO	0x0
8	GPIO18_LEVEL_LOW	RO	0x0
7	GPIO17_EDGE_HIGH	RO	0x0
6	GPIO17_EDGE_LOW	RO	0x0
5	GPIO17_LEVEL_HIGH	RO	0x0
4	GPIO17_LEVEL_LOW	RO	0x0
3	GPIO16_EDGE_HIGH	RO	0x0
2	GPIO16_EDGE_LOW	RO	0x0
1	GPIO16_LEVEL_HIGH	RO	0x0
0	GPIO16_LEVEL_LOW	RO	0x0

IO_BANK0: PROC1_INTS3 Register

Offset: 0x15c

Description

Interrupt status after masking & forcing for proc1

Table 313.
PROC1_INTS3
Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RO	0x0
22	GPIO29_EDGE_LOW	RO	0x0
21	GPIO29_LEVEL_HIGH	RO	0x0
20	GPIO29_LEVEL_LOW	RO	0x0
19	GPIO28_EDGE_HIGH	RO	0x0
18	GPIO28_EDGE_LOW	RO	0x0
17	GPIO28_LEVEL_HIGH	RO	0x0
16	GPIO28_LEVEL_LOW	RO	0x0
15	GPIO27_EDGE_HIGH	RO	0x0
14	GPIO27_EDGE_LOW	RO	0x0
13	GPIO27_LEVEL_HIGH	RO	0x0
12	GPIO27_LEVEL_LOW	RO	0x0
11	GPIO26_EDGE_HIGH	RO	0x0
10	GPIO26_EDGE_LOW	RO	0x0
9	GPIO26_LEVEL_HIGH	RO	0x0
8	GPIO26_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: PROC1_INTS3寄存器

偏移量: 0x15c

说明

proc1 的中断状态 (掩码及强制后)

表313
PROC1_INTS3
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
7	GPIO25_EDGE_HIGH	RO	0x0
6	GPIO25_EDGE_LOW	RO	0x0
5	GPIO25_LEVEL_HIGH	RO	0x0
4	GPIO25_LEVEL_LOW	RO	0x0
3	GPIO24_EDGE_HIGH	RO	0x0
2	GPIO24_EDGE_LOW	RO	0x0
1	GPIO24_LEVEL_HIGH	RO	0x0
0	GPIO24_LEVEL_LOW	RO	0x0

IO_BANK0: DORMANT_WAKE_INTE0 Register

Offset: 0x160

Description

Interrupt Enable for dormant_wake

Table 314.
DORMANT_WAKE_INT
E0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0
10	GPIO2_EDGE_LOW	RW	0x0

位	描述	类型	复位值
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTE0寄存器

偏移量: 0x160

说明

dormant_wake 的中断使能

表314
DORMANT_WAKE_INT
E0寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0
10	GPIO2_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0
7	GPIO1_EDGE_HIGH	RW	0x0
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0
4	GPIO1_LEVEL_LOW	RW	0x0
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTE1 Register

Offset: 0x164

Description

Interrupt Enable for dormant_wake

Table 315.
DORMANT_WAKE_INT
E1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0
12	GPIO11_LEVEL_LOW	RW	0x0

位	描述	类型	复位值
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE1寄存器

偏移量: 0x164

说明

dormant_wake 的中断使能

表315
DORMANT_WAKE_INT
E1寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0
12	GPIO11_LEVEL_LOW	读写	0x0

Bits	Description	Type	Reset
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0
9	GPIO10_LEVEL_HIGH	RW	0x0
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0
6	GPIO9_EDGE_LOW	RW	0x0
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTE2 Register

Offset: 0x168

Description

Interrupt Enable for dormant_wake

Table 316.
DORMANT_WAKE_INT
E2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0
14	GPIO19_EDGE_LOW	RW	0x0

位	描述	类型	复位值
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE2 寄存器

偏移量: 0x168

说明

dormant_wake 的中断使能

表316。
DORMANT_WAKE_INT
E2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0
14	GPIO19_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0
11	GPIO18_EDGE_HIGH	RW	0x0
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0
8	GPIO18_LEVEL_LOW	RW	0x0
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTE3 Register

Offset: 0x16c

Description

Interrupt Enable for dormant_wake

Table 317.
DORMANT_WAKE_INT
E3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0
9	GPIO26_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTE3 寄存器

偏移量: 0x16c

说明

dormant_wake 的中断使能

表317。
DORMANT_WAKE_INT
E3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0
9	GPIO26_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0
6	GPIO25_EDGE_LOW	RW	0x0
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0
3	GPIO24_EDGE_HIGH	RW	0x0
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTF0 Register

Offset: 0x170

Description

Interrupt Force for dormant_wake

Table 318.
DORMANT_WAKE_INT
F0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RW	0x0
30	GPIO7_EDGE_LOW	RW	0x0
29	GPIO7_LEVEL_HIGH	RW	0x0
28	GPIO7_LEVEL_LOW	RW	0x0
27	GPIO6_EDGE_HIGH	RW	0x0
26	GPIO6_EDGE_LOW	RW	0x0
25	GPIO6_LEVEL_HIGH	RW	0x0
24	GPIO6_LEVEL_LOW	RW	0x0
23	GPIO5_EDGE_HIGH	RW	0x0
22	GPIO5_EDGE_LOW	RW	0x0
21	GPIO5_LEVEL_HIGH	RW	0x0
20	GPIO5_LEVEL_LOW	RW	0x0
19	GPIO4_EDGE_HIGH	RW	0x0
18	GPIO4_EDGE_LOW	RW	0x0
17	GPIO4_LEVEL_HIGH	RW	0x0
16	GPIO4_LEVEL_LOW	RW	0x0
15	GPIO3_EDGE_HIGH	RW	0x0
14	GPIO3_EDGE_LOW	RW	0x0
13	GPIO3_LEVEL_HIGH	RW	0x0
12	GPIO3_LEVEL_LOW	RW	0x0
11	GPIO2_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INT0 寄存器

偏移量：0x170

说明

dormant_wake 的中断强制

表318。
DORMANT_WAKE_INT
F0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	读写	0x0
30	GPIO7_EDGE_LOW	读写	0x0
29	GPIO7_LEVEL_HIGH	读写	0x0
28	GPIO7_LEVEL_LOW	读写	0x0
27	GPIO6_EDGE_HIGH	读写	0x0
26	GPIO6_EDGE_LOW	读写	0x0
25	GPIO6_LEVEL_HIGH	读写	0x0
24	GPIO6_LEVEL_LOW	读写	0x0
23	GPIO5_EDGE_HIGH	读写	0x0
22	GPIO5_EDGE_LOW	读写	0x0
21	GPIO5_LEVEL_HIGH	读写	0x0
20	GPIO5_LEVEL_LOW	读写	0x0
19	GPIO4_EDGE_HIGH	读写	0x0
18	GPIO4_EDGE_LOW	读写	0x0
17	GPIO4_LEVEL_HIGH	读写	0x0
16	GPIO4_LEVEL_LOW	读写	0x0
15	GPIO3_EDGE_HIGH	读写	0x0
14	GPIO3_EDGE_LOW	读写	0x0
13	GPIO3_LEVEL_HIGH	读写	0x0
12	GPIO3_LEVEL_LOW	读写	0x0
11	GPIO2_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
10	GPIO2_EDGE_LOW	RW	0x0
9	GPIO2_LEVEL_HIGH	RW	0x0
8	GPIO2_LEVEL_LOW	RW	0x0
7	GPIO1_EDGE_HIGH	RW	0x0
6	GPIO1_EDGE_LOW	RW	0x0
5	GPIO1_LEVEL_HIGH	RW	0x0
4	GPIO1_LEVEL_LOW	RW	0x0
3	GPIO0_EDGE_HIGH	RW	0x0
2	GPIO0_EDGE_LOW	RW	0x0
1	GPIO0_LEVEL_HIGH	RW	0x0
0	GPIO0_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTF1 Register

Offset: 0x174

Description

Interrupt Force for dormant_wake

Table 319.
DORMANT_WAKE_INT
F1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RW	0x0
30	GPIO15_EDGE_LOW	RW	0x0
29	GPIO15_LEVEL_HIGH	RW	0x0
28	GPIO15_LEVEL_LOW	RW	0x0
27	GPIO14_EDGE_HIGH	RW	0x0
26	GPIO14_EDGE_LOW	RW	0x0
25	GPIO14_LEVEL_HIGH	RW	0x0
24	GPIO14_LEVEL_LOW	RW	0x0
23	GPIO13_EDGE_HIGH	RW	0x0
22	GPIO13_EDGE_LOW	RW	0x0
21	GPIO13_LEVEL_HIGH	RW	0x0
20	GPIO13_LEVEL_LOW	RW	0x0
19	GPIO12_EDGE_HIGH	RW	0x0
18	GPIO12_EDGE_LOW	RW	0x0
17	GPIO12_LEVEL_HIGH	RW	0x0
16	GPIO12_LEVEL_LOW	RW	0x0
15	GPIO11_EDGE_HIGH	RW	0x0
14	GPIO11_EDGE_LOW	RW	0x0
13	GPIO11_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
10	GPIO2_EDGE_LOW	读写	0x0
9	GPIO2_LEVEL_HIGH	读写	0x0
8	GPIO2_LEVEL_LOW	读写	0x0
7	GPIO1_EDGE_HIGH	读写	0x0
6	GPIO1_EDGE_LOW	读写	0x0
5	GPIO1_LEVEL_HIGH	读写	0x0
4	GPIO1_LEVEL_LOW	读写	0x0
3	GPIO0_EDGE_HIGH	读写	0x0
2	GPIO0_EDGE_LOW	读写	0x0
1	GPIO0_LEVEL_HIGH	读写	0x0
0	GPIO0_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF1 寄存器

偏移量: 0x174

说明

dormant_wake 的中断强制

表319。
DORMANT_WAKE_INT
F1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	读写	0x0
30	GPIO15_EDGE_LOW	读写	0x0
29	GPIO15_LEVEL_HIGH	读写	0x0
28	GPIO15_LEVEL_LOW	读写	0x0
27	GPIO14_EDGE_HIGH	读写	0x0
26	GPIO14_EDGE_LOW	读写	0x0
25	GPIO14_LEVEL_HIGH	读写	0x0
24	GPIO14_LEVEL_LOW	读写	0x0
23	GPIO13_EDGE_HIGH	读写	0x0
22	GPIO13_EDGE_LOW	读写	0x0
21	GPIO13_LEVEL_HIGH	读写	0x0
20	GPIO13_LEVEL_LOW	读写	0x0
19	GPIO12_EDGE_HIGH	读写	0x0
18	GPIO12_EDGE_LOW	读写	0x0
17	GPIO12_LEVEL_HIGH	读写	0x0
16	GPIO12_LEVEL_LOW	读写	0x0
15	GPIO11_EDGE_HIGH	读写	0x0
14	GPIO11_EDGE_LOW	读写	0x0
13	GPIO11_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
12	GPIO11_LEVEL_LOW	RW	0x0
11	GPIO10_EDGE_HIGH	RW	0x0
10	GPIO10_EDGE_LOW	RW	0x0
9	GPIO10_LEVEL_HIGH	RW	0x0
8	GPIO10_LEVEL_LOW	RW	0x0
7	GPIO9_EDGE_HIGH	RW	0x0
6	GPIO9_EDGE_LOW	RW	0x0
5	GPIO9_LEVEL_HIGH	RW	0x0
4	GPIO9_LEVEL_LOW	RW	0x0
3	GPIO8_EDGE_HIGH	RW	0x0
2	GPIO8_EDGE_LOW	RW	0x0
1	GPIO8_LEVEL_HIGH	RW	0x0
0	GPIO8_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTF2 Register

Offset: 0x178

Description

Interrupt Force for dormant_wake

Table 320.
DORMANT_WAKE_INT
F2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RW	0x0
30	GPIO23_EDGE_LOW	RW	0x0
29	GPIO23_LEVEL_HIGH	RW	0x0
28	GPIO23_LEVEL_LOW	RW	0x0
27	GPIO22_EDGE_HIGH	RW	0x0
26	GPIO22_EDGE_LOW	RW	0x0
25	GPIO22_LEVEL_HIGH	RW	0x0
24	GPIO22_LEVEL_LOW	RW	0x0
23	GPIO21_EDGE_HIGH	RW	0x0
22	GPIO21_EDGE_LOW	RW	0x0
21	GPIO21_LEVEL_HIGH	RW	0x0
20	GPIO21_LEVEL_LOW	RW	0x0
19	GPIO20_EDGE_HIGH	RW	0x0
18	GPIO20_EDGE_LOW	RW	0x0
17	GPIO20_LEVEL_HIGH	RW	0x0
16	GPIO20_LEVEL_LOW	RW	0x0
15	GPIO19_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
12	GPIO11_LEVEL_LOW	读写	0x0
11	GPIO10_EDGE_HIGH	读写	0x0
10	GPIO10_EDGE_LOW	读写	0x0
9	GPIO10_LEVEL_HIGH	读写	0x0
8	GPIO10_LEVEL_LOW	读写	0x0
7	GPIO9_EDGE_HIGH	读写	0x0
6	GPIO9_EDGE_LOW	读写	0x0
5	GPIO9_LEVEL_HIGH	读写	0x0
4	GPIO9_LEVEL_LOW	读写	0x0
3	GPIO8_EDGE_HIGH	读写	0x0
2	GPIO8_EDGE_LOW	读写	0x0
1	GPIO8_LEVEL_HIGH	读写	0x0
0	GPIO8_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF2 寄存器

偏移量: 0x178

说明

dormant_wake 的中断强制

表 320。
DORMANT_WAKE_INTF2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	读写	0x0
30	GPIO23_EDGE_LOW	读写	0x0
29	GPIO23_LEVEL_HIGH	读写	0x0
28	GPIO23_LEVEL_LOW	读写	0x0
27	GPIO22_EDGE_HIGH	读写	0x0
26	GPIO22_EDGE_LOW	读写	0x0
25	GPIO22_LEVEL_HIGH	读写	0x0
24	GPIO22_LEVEL_LOW	读写	0x0
23	GPIO21_EDGE_HIGH	读写	0x0
22	GPIO21_EDGE_LOW	读写	0x0
21	GPIO21_LEVEL_HIGH	读写	0x0
20	GPIO21_LEVEL_LOW	读写	0x0
19	GPIO20_EDGE_HIGH	读写	0x0
18	GPIO20_EDGE_LOW	读写	0x0
17	GPIO20_LEVEL_HIGH	读写	0x0
16	GPIO20_LEVEL_LOW	读写	0x0
15	GPIO19_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
14	GPIO19_EDGE_LOW	RW	0x0
13	GPIO19_LEVEL_HIGH	RW	0x0
12	GPIO19_LEVEL_LOW	RW	0x0
11	GPIO18_EDGE_HIGH	RW	0x0
10	GPIO18_EDGE_LOW	RW	0x0
9	GPIO18_LEVEL_HIGH	RW	0x0
8	GPIO18_LEVEL_LOW	RW	0x0
7	GPIO17_EDGE_HIGH	RW	0x0
6	GPIO17_EDGE_LOW	RW	0x0
5	GPIO17_LEVEL_HIGH	RW	0x0
4	GPIO17_LEVEL_LOW	RW	0x0
3	GPIO16_EDGE_HIGH	RW	0x0
2	GPIO16_EDGE_LOW	RW	0x0
1	GPIO16_LEVEL_HIGH	RW	0x0
0	GPIO16_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTF3 Register

Offset: 0x17c

Description

Interrupt Force for dormant_wake

Table 321.
DORMANT_WAKE_INT
F3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RW	0x0
22	GPIO29_EDGE_LOW	RW	0x0
21	GPIO29_LEVEL_HIGH	RW	0x0
20	GPIO29_LEVEL_LOW	RW	0x0
19	GPIO28_EDGE_HIGH	RW	0x0
18	GPIO28_EDGE_LOW	RW	0x0
17	GPIO28_LEVEL_HIGH	RW	0x0
16	GPIO28_LEVEL_LOW	RW	0x0
15	GPIO27_EDGE_HIGH	RW	0x0
14	GPIO27_EDGE_LOW	RW	0x0
13	GPIO27_LEVEL_HIGH	RW	0x0
12	GPIO27_LEVEL_LOW	RW	0x0
11	GPIO26_EDGE_HIGH	RW	0x0
10	GPIO26_EDGE_LOW	RW	0x0

位	描述	类型	复位值
14	GPIO19_EDGE_LOW	读写	0x0
13	GPIO19_LEVEL_HIGH	读写	0x0
12	GPIO19_LEVEL_LOW	读写	0x0
11	GPIO18_EDGE_HIGH	读写	0x0
10	GPIO18_EDGE_LOW	读写	0x0
9	GPIO18_LEVEL_HIGH	读写	0x0
8	GPIO18_LEVEL_LOW	读写	0x0
7	GPIO17_EDGE_HIGH	读写	0x0
6	GPIO17_EDGE_LOW	读写	0x0
5	GPIO17_LEVEL_HIGH	读写	0x0
4	GPIO17_LEVEL_LOW	读写	0x0
3	GPIO16_EDGE_HIGH	读写	0x0
2	GPIO16_EDGE_LOW	读写	0x0
1	GPIO16_LEVEL_HIGH	读写	0x0
0	GPIO16_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTF3 寄存器

偏移量: 0x17c

描述

dormant_wake 的中断强制

表 321。
DORMANT_WAKE_INT
F3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	读写	0x0
22	GPIO29_EDGE_LOW	读写	0x0
21	GPIO29_LEVEL_HIGH	读写	0x0
20	GPIO29_LEVEL_LOW	读写	0x0
19	GPIO28_EDGE_HIGH	读写	0x0
18	GPIO28_EDGE_LOW	读写	0x0
17	GPIO28_LEVEL_HIGH	读写	0x0
16	GPIO28_LEVEL_LOW	读写	0x0
15	GPIO27_EDGE_HIGH	读写	0x0
14	GPIO27_EDGE_LOW	读写	0x0
13	GPIO27_LEVEL_HIGH	读写	0x0
12	GPIO27_LEVEL_LOW	读写	0x0
11	GPIO26_EDGE_HIGH	读写	0x0
10	GPIO26_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
9	GPIO26_LEVEL_HIGH	RW	0x0
8	GPIO26_LEVEL_LOW	RW	0x0
7	GPIO25_EDGE_HIGH	RW	0x0
6	GPIO25_EDGE_LOW	RW	0x0
5	GPIO25_LEVEL_HIGH	RW	0x0
4	GPIO25_LEVEL_LOW	RW	0x0
3	GPIO24_EDGE_HIGH	RW	0x0
2	GPIO24_EDGE_LOW	RW	0x0
1	GPIO24_LEVEL_HIGH	RW	0x0
0	GPIO24_LEVEL_LOW	RW	0x0

IO_BANK0: DORMANT_WAKE_INTS0 Register

Offset: 0x180

Description

Interrupt status after masking & forcing for dormant_wake

Table 322.
DORMANT_WAKE_INT_S0 Register

Bits	Description	Type	Reset
31	GPIO7_EDGE_HIGH	RO	0x0
30	GPIO7_EDGE_LOW	RO	0x0
29	GPIO7_LEVEL_HIGH	RO	0x0
28	GPIO7_LEVEL_LOW	RO	0x0
27	GPIO6_EDGE_HIGH	RO	0x0
26	GPIO6_EDGE_LOW	RO	0x0
25	GPIO6_LEVEL_HIGH	RO	0x0
24	GPIO6_LEVEL_LOW	RO	0x0
23	GPIO5_EDGE_HIGH	RO	0x0
22	GPIO5_EDGE_LOW	RO	0x0
21	GPIO5_LEVEL_HIGH	RO	0x0
20	GPIO5_LEVEL_LOW	RO	0x0
19	GPIO4_EDGE_HIGH	RO	0x0
18	GPIO4_EDGE_LOW	RO	0x0
17	GPIO4_LEVEL_HIGH	RO	0x0
16	GPIO4_LEVEL_LOW	RO	0x0
15	GPIO3_EDGE_HIGH	RO	0x0
14	GPIO3_EDGE_LOW	RO	0x0
13	GPIO3_LEVEL_HIGH	RO	0x0
12	GPIO3_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
9	GPIO26_LEVEL_HIGH	读写	0x0
8	GPIO26_LEVEL_LOW	读写	0x0
7	GPIO25_EDGE_HIGH	读写	0x0
6	GPIO25_EDGE_LOW	读写	0x0
5	GPIO25_LEVEL_HIGH	读写	0x0
4	GPIO25_LEVEL_LOW	读写	0x0
3	GPIO24_EDGE_HIGH	读写	0x0
2	GPIO24_EDGE_LOW	读写	0x0
1	GPIO24_LEVEL_HIGH	读写	0x0
0	GPIO24_LEVEL_LOW	读写	0x0

IO_BANK0: DORMANT_WAKE_INTS0 寄存器

偏移量: 0x180

描述

dormant_wake 遮蔽及强制后的中断状态

表 322。
DORMANT_WAKE_INTS0 寄存器

位	描述	类型	复位值
31	GPIO7_EDGE_HIGH	只读	0x0
30	GPIO7_EDGE_LOW	只读	0x0
29	GPIO7_LEVEL_HIGH	只读	0x0
28	GPIO7_LEVEL_LOW	只读	0x0
27	GPIO6_EDGE_HIGH	只读	0x0
26	GPIO6_EDGE_LOW	只读	0x0
25	GPIO6_LEVEL_HIGH	只读	0x0
24	GPIO6_LEVEL_LOW	只读	0x0
23	GPIO5_EDGE_HIGH	只读	0x0
22	GPIO5_EDGE_LOW	只读	0x0
21	GPIO5_LEVEL_HIGH	只读	0x0
20	GPIO5_LEVEL_LOW	只读	0x0
19	GPIO4_EDGE_HIGH	只读	0x0
18	GPIO4_EDGE_LOW	只读	0x0
17	GPIO4_LEVEL_HIGH	只读	0x0
16	GPIO4_LEVEL_LOW	只读	0x0
15	GPIO3_EDGE_HIGH	只读	0x0
14	GPIO3_EDGE_LOW	只读	0x0
13	GPIO3_LEVEL_HIGH	只读	0x0
12	GPIO3_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
11	GPIO2_EDGE_HIGH	RO	0x0
10	GPIO2_EDGE_LOW	RO	0x0
9	GPIO2_LEVEL_HIGH	RO	0x0
8	GPIO2_LEVEL_LOW	RO	0x0
7	GPIO1_EDGE_HIGH	RO	0x0
6	GPIO1_EDGE_LOW	RO	0x0
5	GPIO1_LEVEL_HIGH	RO	0x0
4	GPIO1_LEVEL_LOW	RO	0x0
3	GPIO0_EDGE_HIGH	RO	0x0
2	GPIO0_EDGE_LOW	RO	0x0
1	GPIO0_LEVEL_HIGH	RO	0x0
0	GPIO0_LEVEL_LOW	RO	0x0

IO_BANK0: DORMANT_WAKE_INTS1 Register

Offset: 0x184

Description

Interrupt status after masking & forcing for dormant_wake

Table 323.
DORMANT_WAKE_INT
S1 Register

Bits	Description	Type	Reset
31	GPIO15_EDGE_HIGH	RO	0x0
30	GPIO15_EDGE_LOW	RO	0x0
29	GPIO15_LEVEL_HIGH	RO	0x0
28	GPIO15_LEVEL_LOW	RO	0x0
27	GPIO14_EDGE_HIGH	RO	0x0
26	GPIO14_EDGE_LOW	RO	0x0
25	GPIO14_LEVEL_HIGH	RO	0x0
24	GPIO14_LEVEL_LOW	RO	0x0
23	GPIO13_EDGE_HIGH	RO	0x0
22	GPIO13_EDGE_LOW	RO	0x0
21	GPIO13_LEVEL_HIGH	RO	0x0
20	GPIO13_LEVEL_LOW	RO	0x0
19	GPIO12_EDGE_HIGH	RO	0x0
18	GPIO12_EDGE_LOW	RO	0x0
17	GPIO12_LEVEL_HIGH	RO	0x0
16	GPIO12_LEVEL_LOW	RO	0x0
15	GPIO11_EDGE_HIGH	RO	0x0
14	GPIO11_EDGE_LOW	RO	0x0

位	描述	类型	复位值
11	GPIO2_EDGE_HIGH	只读	0x0
10	GPIO2_EDGE_LOW	只读	0x0
9	GPIO2_LEVEL_HIGH	只读	0x0
8	GPIO2_LEVEL_LOW	只读	0x0
7	GPIO1_EDGE_HIGH	只读	0x0
6	GPIO1_EDGE_LOW	只读	0x0
5	GPIO1_LEVEL_HIGH	只读	0x0
4	GPIO1_LEVEL_LOW	只读	0x0
3	GPIO0_EDGE_HIGH	只读	0x0
2	GPIO0_EDGE_LOW	只读	0x0
1	GPIO0_LEVEL_HIGH	只读	0x0
0	GPIO0_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS1 寄存器

偏移量: 0x184

描述

dormant_wake 遮蔽及强制后的中断状态

表 323。
DORMANT_WAKE_INT
S1 寄存器

位	描述	类型	复位值
31	GPIO15_EDGE_HIGH	只读	0x0
30	GPIO15_EDGE_LOW	只读	0x0
29	GPIO15_LEVEL_HIGH	只读	0x0
28	GPIO15_LEVEL_LOW	只读	0x0
27	GPIO14_EDGE_HIGH	只读	0x0
26	GPIO14_EDGE_LOW	只读	0x0
25	GPIO14_LEVEL_HIGH	只读	0x0
24	GPIO14_LEVEL_LOW	只读	0x0
23	GPIO13_EDGE_HIGH	只读	0x0
22	GPIO13_EDGE_LOW	只读	0x0
21	GPIO13_LEVEL_HIGH	只读	0x0
20	GPIO13_LEVEL_LOW	只读	0x0
19	GPIO12_EDGE_HIGH	只读	0x0
18	GPIO12_EDGE_LOW	只读	0x0
17	GPIO12_LEVEL_HIGH	只读	0x0
16	GPIO12_LEVEL_LOW	只读	0x0
15	GPIO11_EDGE_HIGH	只读	0x0
14	GPIO11_EDGE_LOW	只读	0x0

Bits	Description	Type	Reset
13	GPIO11_LEVEL_HIGH	RO	0x0
12	GPIO11_LEVEL_LOW	RO	0x0
11	GPIO10_EDGE_HIGH	RO	0x0
10	GPIO10_EDGE_LOW	RO	0x0
9	GPIO10_LEVEL_HIGH	RO	0x0
8	GPIO10_LEVEL_LOW	RO	0x0
7	GPIO9_EDGE_HIGH	RO	0x0
6	GPIO9_EDGE_LOW	RO	0x0
5	GPIO9_LEVEL_HIGH	RO	0x0
4	GPIO9_LEVEL_LOW	RO	0x0
3	GPIO8_EDGE_HIGH	RO	0x0
2	GPIO8_EDGE_LOW	RO	0x0
1	GPIO8_LEVEL_HIGH	RO	0x0
0	GPIO8_LEVEL_LOW	RO	0x0

IO_BANK0: DORMANT_WAKE_INTS2 Register

Offset: 0x188

Description

Interrupt status after masking & forcing for dormant_wake

Table 324.
DORMANT_WAKE_INT
S2 Register

Bits	Description	Type	Reset
31	GPIO23_EDGE_HIGH	RO	0x0
30	GPIO23_EDGE_LOW	RO	0x0
29	GPIO23_LEVEL_HIGH	RO	0x0
28	GPIO23_LEVEL_LOW	RO	0x0
27	GPIO22_EDGE_HIGH	RO	0x0
26	GPIO22_EDGE_LOW	RO	0x0
25	GPIO22_LEVEL_HIGH	RO	0x0
24	GPIO22_LEVEL_LOW	RO	0x0
23	GPIO21_EDGE_HIGH	RO	0x0
22	GPIO21_EDGE_LOW	RO	0x0
21	GPIO21_LEVEL_HIGH	RO	0x0
20	GPIO21_LEVEL_LOW	RO	0x0
19	GPIO20_EDGE_HIGH	RO	0x0
18	GPIO20_EDGE_LOW	RO	0x0
17	GPIO20_LEVEL_HIGH	RO	0x0
16	GPIO20_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
13	GPIO11_LEVEL_HIGH	只读	0x0
12	GPIO11_LEVEL_LOW	只读	0x0
11	GPIO10_EDGE_HIGH	只读	0x0
10	GPIO10_EDGE_LOW	只读	0x0
9	GPIO10_LEVEL_HIGH	只读	0x0
8	GPIO10_LEVEL_LOW	只读	0x0
7	GPIO9_EDGE_HIGH	只读	0x0
6	GPIO9_EDGE_LOW	只读	0x0
5	GPIO9_LEVEL_HIGH	只读	0x0
4	GPIO9_LEVEL_LOW	只读	0x0
3	GPIO8_EDGE_HIGH	只读	0x0
2	GPIO8_EDGE_LOW	只读	0x0
1	GPIO8_LEVEL_HIGH	只读	0x0
0	GPIO8_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS2 寄存器

偏移量: 0x188

描述

dormant_wake 遮蔽及强制后的中断状态

表 324。
DORMANT_WAKE_INT
S2 寄存器

位	描述	类型	复位值
31	GPIO23_EDGE_HIGH	只读	0x0
30	GPIO23_EDGE_LOW	只读	0x0
29	GPIO23_LEVEL_HIGH	只读	0x0
28	GPIO23_LEVEL_LOW	只读	0x0
27	GPIO22_EDGE_HIGH	只读	0x0
26	GPIO22_EDGE_LOW	只读	0x0
25	GPIO22_LEVEL_HIGH	只读	0x0
24	GPIO22_LEVEL_LOW	只读	0x0
23	GPIO21_EDGE_HIGH	只读	0x0
22	GPIO21_EDGE_LOW	只读	0x0
21	GPIO21_LEVEL_HIGH	只读	0x0
20	GPIO21_LEVEL_LOW	只读	0x0
19	GPIO20_EDGE_HIGH	只读	0x0
18	GPIO20_EDGE_LOW	只读	0x0
17	GPIO20_LEVEL_HIGH	只读	0x0
16	GPIO20_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
15	GPIO19_EDGE_HIGH	RO	0x0
14	GPIO19_EDGE_LOW	RO	0x0
13	GPIO19_LEVEL_HIGH	RO	0x0
12	GPIO19_LEVEL_LOW	RO	0x0
11	GPIO18_EDGE_HIGH	RO	0x0
10	GPIO18_EDGE_LOW	RO	0x0
9	GPIO18_LEVEL_HIGH	RO	0x0
8	GPIO18_LEVEL_LOW	RO	0x0
7	GPIO17_EDGE_HIGH	RO	0x0
6	GPIO17_EDGE_LOW	RO	0x0
5	GPIO17_LEVEL_HIGH	RO	0x0
4	GPIO17_LEVEL_LOW	RO	0x0
3	GPIO16_EDGE_HIGH	RO	0x0
2	GPIO16_EDGE_LOW	RO	0x0
1	GPIO16_LEVEL_HIGH	RO	0x0
0	GPIO16_LEVEL_LOW	RO	0x0

IO_BANK0: DORMANT_WAKE_INTS3 Register

Offset: 0x18c

Description

Interrupt status after masking & forcing for dormant_wake

Table 325.
DORMANT_WAKE_INT
S3 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO29_EDGE_HIGH	RO	0x0
22	GPIO29_EDGE_LOW	RO	0x0
21	GPIO29_LEVEL_HIGH	RO	0x0
20	GPIO29_LEVEL_LOW	RO	0x0
19	GPIO28_EDGE_HIGH	RO	0x0
18	GPIO28_EDGE_LOW	RO	0x0
17	GPIO28_LEVEL_HIGH	RO	0x0
16	GPIO28_LEVEL_LOW	RO	0x0
15	GPIO27_EDGE_HIGH	RO	0x0
14	GPIO27_EDGE_LOW	RO	0x0
13	GPIO27_LEVEL_HIGH	RO	0x0
12	GPIO27_LEVEL_LOW	RO	0x0
11	GPIO26_EDGE_HIGH	RO	0x0

位	描述	类型	复位值
15	GPIO19_EDGE_HIGH	只读	0x0
14	GPIO19_EDGE_LOW	只读	0x0
13	GPIO19_LEVEL_HIGH	只读	0x0
12	GPIO19_LEVEL_LOW	只读	0x0
11	GPIO18_EDGE_HIGH	只读	0x0
10	GPIO18_EDGE_LOW	只读	0x0
9	GPIO18_LEVEL_HIGH	只读	0x0
8	GPIO18_LEVEL_LOW	只读	0x0
7	GPIO17_EDGE_HIGH	只读	0x0
6	GPIO17_EDGE_LOW	只读	0x0
5	GPIO17_LEVEL_HIGH	只读	0x0
4	GPIO17_LEVEL_LOW	只读	0x0
3	GPIO16_EDGE_HIGH	只读	0x0
2	GPIO16_EDGE_LOW	只读	0x0
1	GPIO16_LEVEL_HIGH	只读	0x0
0	GPIO16_LEVEL_LOW	只读	0x0

IO_BANK0: DORMANT_WAKE_INTS3 寄存器

偏移量: 0x18c

描述

dormant_wake 遮蔽及强制后的中断状态

表 325。
DORMANT_WAKE_INT
S3 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO29_EDGE_HIGH	只读	0x0
22	GPIO29_EDGE_LOW	只读	0x0
21	GPIO29_LEVEL_HIGH	只读	0x0
20	GPIO29_LEVEL_LOW	只读	0x0
19	GPIO28_EDGE_HIGH	只读	0x0
18	GPIO28_EDGE_LOW	只读	0x0
17	GPIO28_LEVEL_HIGH	只读	0x0
16	GPIO28_LEVEL_LOW	只读	0x0
15	GPIO27_EDGE_HIGH	只读	0x0
14	GPIO27_EDGE_LOW	只读	0x0
13	GPIO27_LEVEL_HIGH	只读	0x0
12	GPIO27_LEVEL_LOW	只读	0x0
11	GPIO26_EDGE_HIGH	只读	0x0

Bits	Description	Type	Reset
10	GPIO26_EDGE_LOW	RO	0x0
9	GPIO26_LEVEL_HIGH	RO	0x0
8	GPIO26_LEVEL_LOW	RO	0x0
7	GPIO25_EDGE_HIGH	RO	0x0
6	GPIO25_EDGE_LOW	RO	0x0
5	GPIO25_LEVEL_HIGH	RO	0x0
4	GPIO25_LEVEL_LOW	RO	0x0
3	GPIO24_EDGE_HIGH	RO	0x0
2	GPIO24_EDGE_LOW	RO	0x0
1	GPIO24_LEVEL_HIGH	RO	0x0
0	GPIO24_LEVEL_LOW	RO	0x0

2.19.6.2. IO - QSPI Bank

The QSPI Bank IO registers start at a base address of **0x40018000** (defined as **IO_QSPI_BASE** in SDK).

Table 326. List of
IO_QSPI registers

Offset	Name	Info
0x00	GPIO_QSPI_SCLK_STATUS	GPIO status
0x04	GPIO_QSPI_SCLK_CTRL	GPIO control including function select and overrides.
0x08	GPIO_QSPI_SS_STATUS	GPIO status
0x0c	GPIO_QSPI_SS_CTRL	GPIO control including function select and overrides.
0x10	GPIO_QSPI_SD0_STATUS	GPIO status
0x14	GPIO_QSPI_SD0_CTRL	GPIO control including function select and overrides.
0x18	GPIO_QSPI_SD1_STATUS	GPIO status
0x1c	GPIO_QSPI_SD1_CTRL	GPIO control including function select and overrides.
0x20	GPIO_QSPI_SD2_STATUS	GPIO status
0x24	GPIO_QSPI_SD2_CTRL	GPIO control including function select and overrides.
0x28	GPIO_QSPI_SD3_STATUS	GPIO status
0x2c	GPIO_QSPI_SD3_CTRL	GPIO control including function select and overrides.
0x30	INTR	Raw Interrupts
0x34	PROC0_INTE	Interrupt Enable for proc0
0x38	PROC0_INTF	Interrupt Force for proc0
0x3c	PROC0_INTS	Interrupt status after masking & forcing for proc0
0x40	PROC1_INTE	Interrupt Enable for proc1
0x44	PROC1_INTF	Interrupt Force for proc1
0x48	PROC1_INTS	Interrupt status after masking & forcing for proc1
0x4c	DORMANT_WAKE_INTE	Interrupt Enable for dormant_wake

位	描述	类型	复位值
10	GPIO26_EDGE_LOW	只读	0x0
9	GPIO26_LEVEL_HIGH	只读	0x0
8	GPIO26_LEVEL_LOW	只读	0x0
7	GPIO25_EDGE_HIGH	只读	0x0
6	GPIO25_EDGE_LOW	只读	0x0
5	GPIO25_LEVEL_HIGH	只读	0x0
4	GPIO25_LEVEL_LOW	只读	0x0
3	GPIO24_EDGE_HIGH	只读	0x0
2	GPIO24_EDGE_LOW	只读	0x0
1	GPIO24_LEVEL_HIGH	只读	0x0
0	GPIO24_LEVEL_LOW	只读	0x0

2.19.6.2. IO - QSPI 银行

QSPI 银行 IO 寄存器起始基地址为 **0x40018000** (在 SDK 中定义为 **IO_QSPI_BASE**)。

表 326。IO_QSPI
寄存器列表

偏移量	名称	说明
0x00	GPIO_QSPI_SCLK_STATUS	GPIO 状态
0x04	GPIO_QSPI_SCLK_CTRL	GPIO 控制，包括功能选择和覆盖。
0x08	GPIO_QSPI_SS_STATUS	GPIO 状态
0x0c	GPIO_QSPI_SS_CTRL	GPIO 控制，包括功能选择和覆盖。
0x10	GPIO_QSPI_SD0_STATUS	GPIO 状态
0x14	GPIO_QSPI_SD0_CTRL	GPIO 控制，包括功能选择和覆盖。
0x18	GPIO_QSPI_SD1_STATUS	GPIO 状态
0x1c	GPIO_QSPI_SD1_CTRL	GPIO 控制，包括功能选择和覆盖。
0x20	GPIO_QSPI_SD2_STATUS	GPIO 状态
0x24	GPIO_QSPI_SD2_CTRL	GPIO 控制，包括功能选择和覆盖。
0x28	GPIO_QSPI_SD3_STATUS	GPIO 状态
0x2c	GPIO_QSPI_SD3_CTRL	GPIO 控制，包括功能选择和覆盖。
0x30	INTR	原始中断
0x34	PROC0_INTE	proc0 的中断使能
0x38	PROC0_INTF	proc0 的中断强制
0x3c	PROC0_INTS	proc0 的中断状态 (掩码及强制后)
0x40	PROC1_INTE	proc1 的中断使能
0x44	PROC1_INTF	proc1 的中断强制
0x48	PROC1_INTS	proc1 的中断状态 (掩码及强制后)
0x4c	DORMANT_WAKE_INTE	dormant_wake 的中断使能

Offset	Name	Info
0x50	DORMANT_WAKE_INTF	Interrupt Force for dormant_wake
0x54	DORMANT_WAKE_INTS	Interrupt status after masking & forcing for dormant_wake

IO_QSPI: GPIO_QSPI_SCLK_STATUS, GPIO_QSPI_SS_STATUS, ..., GPIO_QSPI_SD2_STATUS, GPIO_QSPI_SD3_STATUS Registers

Offsets: 0x00, 0x08, ..., 0x20, 0x28

Description

GPIO status

Table 327.
GPIO_QSPI_SCLK_STA
TUS,
GPIO_QSPI_SS_STATU
S, ...
GPIO_QSPI_SD2_STAT
US,
GPIO_QSPI_SD3_STAT
US Registers

Bits	Description	Type	Reset
31:27	Reserved.	-	-
26	IRQTOPROC : interrupt to processors, after override is applied	RO	0x0
25	Reserved.	-	-
24	IRQFROMPAD : interrupt from pad before override is applied	RO	0x0
23:20	Reserved.	-	-
19	INTOPERI : input signal to peripheral, after override is applied	RO	0x0
18	Reserved.	-	-
17	INFROMPAD : input signal from pad, before override is applied	RO	0x0
16:14	Reserved.	-	-
13	OETOPAD : output enable to pad after register override is applied	RO	0x0
12	OEFROMPERI : output enable from selected peripheral, before register override is applied	RO	0x0
11:10	Reserved.	-	-
9	OUTTOPAD : output signal to pad after register override is applied	RO	0x0
8	OUTFROMPERI : output signal from selected peripheral, before register override is applied	RO	0x0
7:0	Reserved.	-	-

IO_QSPI: GPIO_QSPI_SCLK_CTRL, GPIO_QSPI_SS_CTRL, ..., GPIO_QSPI_SD2_CTRL, GPIO_QSPI_SD3_CTRL Registers

Offsets: 0x04, 0x0c, ..., 0x24, 0x2c

Description

GPIO control including function select and overrides.

Table 328.
GPIO_QSPI_SCLK_CTR
L,
GPIO_QSPI_SS_CTRL,
...
GPIO_QSPI_SD2_CTRL,
GPIO_QSPI_SD3_CTRL
Registers

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:28	IRQOVER	RW	0x0
	Enumerated values: 0x0 → NORMAL: don't invert the interrupt		

偏移量	名称	说明
0x50	DORMANT_WAKE_INTF	dormant_wake 的中断强制
0x54	DORMANT_WAKE_INTS	dormant_wake 遮蔽及强制后的中断状态

IO_QSPI: GPIO_QSPI_SCLK_STATUS, GPIO_QSPI_SS_STATUS, ..., GPIO_QSPI_SD2_STATUS, GPIO_QSPI_SD3_STATUS 寄存器

偏移量: 0x00, 0x08, ..., 0x20, 0x28

描述

GPIO 状态

表 327。
GPIO_QSPI_SCLK_STA
TUS,
GPIO_QSPI_SS_STATU
S, ...
GPIO_QSPI_SD2_STAT
US,
GPIO_QSPI_SD3_STA
TUS 寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	IRQTOPROC : 应用覆盖后传递给处理器的中断	只读	0x0
25	保留。	-	-
24	IRQFROMPAD : 应用覆盖前来自引脚的中断	只读	0x0
23:20	保留。	-	-
19	INTOPERI : 应用覆盖后传入外设的信号	只读	0x0
18	保留。	-	-
17	INFROMPAD : 应用覆盖前来自引脚的输入信号	只读	0x0
16:14	保留。	-	-
13	OETOPAD : 应用寄存器覆盖后传给引脚的输出使能	只读	0x0
12	OEFROMPERI : 应用寄存器覆盖前来自选定外设的输出使能	只读	0x0
11:10	保留。	-	-
9	OUTTOPAD : 应用寄存器覆盖后传给引脚的输出信号	只读	0x0
8	OUTFROMPERI : 所选外设的输出信号，在寄存器覆盖生效前	只读	0x0
7:0	保留。	-	-

IO_QSPI: GPIO_QSPI_SCLK_CTRL, GPIO_QSPI_SS_CTRL, ..., GPIO_QSPI_SD2_CTRL, GPIO_QSPI_SD3_CTRL 寄存器

偏移量: 0x04, 0x0c, ..., 0x24, 0

x2c 说明

GPIO 控制，包括功能选择和覆盖。

表328。
GPIO_QSPI_SCLK_CTR
L,
GPIO_QSPI_SS_CTRL,
...
GPIO_QSPI_SD2_CTRL,
GPIO_QSPI_SD3_CTRL
寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:28	IRQOVER	读写	0x0
	枚举值: 0x0 → NORMAL: 中断不反转		

Bits	Description	Type	Reset
	0x1 → INVERT: invert the interrupt		
	0x2 → LOW: drive interrupt low		
	0x3 → HIGH: drive interrupt high		
27:18	Reserved.	-	-
17:16	INOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: don't invert the peri input		
	0x1 → INVERT: invert the peri input		
	0x2 → LOW: drive peri input low		
	0x3 → HIGH: drive peri input high		
15:14	Reserved.	-	-
13:12	OEOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: drive output enable from peripheral signal selected by funcsel		
	0x1 → INVERT: drive output enable from inverse of peripheral signal selected by funcsel		
	0x2 → DISABLE: disable output		
	0x3 → ENABLE: enable output		
11:10	Reserved.	-	-
9:8	OUTOVER	RW	0x0
	Enumerated values:		
	0x0 → NORMAL: drive output from peripheral signal selected by funcsel		
	0x1 → INVERT: drive output from inverse of peripheral signal selected by funcsel		
	0x2 → LOW: drive output low		
	0x3 → HIGH: drive output high		
7:5	Reserved.	-	-
4:0	FUNCSEL : Function select. 31 == NULL. See GPIO function table for available functions.	RW	0x1f

IO_QSPI: INTR Register

Offset: 0x30

Description

Raw Interrupts

Table 329. INTR Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	WC	0x0

位	描述	类型	复位值
	0x1 → INVERT：中断反转		
	0x2 → LOW：中断拉低		
	0x3 → HIGH：中断拉高		
27:18	保留。	-	-
17:16	INOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：外设输入不反转		
	0x1 → INVERT：外设输入反转		
	0x2 → LOW：外设输入拉低		
	0x3 → HIGH：外设输入拉高		
15:14	保留。	-	-
13:12	OEOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出使能，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出使能，由funcsel选择的外设信号的反相信号控制		
	0x2 → DISABLE：禁用输出		
	0x3 → ENABLE：使能输出		
11:10	保留。	-	-
9:8	OUTOVER	读写	0x0
	枚举值：		
	0x0 → NORMAL：驱动输出，由funcsel选择的外设信号控制		
	0x1 → INVERT：驱动输出，由funcsel选择的外设信号的反相信号控制		
	0x2 → LOW：驱动输出低电平		
	0x3 → HIGH：驱动输出高电平		
7:5	保留。	-	-
4:0	FUNCSEL ：功能选择。31表示NULL。具体可用功能请参见GPIO功能表。	读写	0x1f

IO_QSPI: INTR 寄存器

偏移量: 0x30

说明

原始中断

表 329. INTR 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	WC	0x0

Bits	Description	Type	Reset
22	GPIO_QSPI_SD3_EDGE_LOW	WC	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RO	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RO	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	WC	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	WC	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RO	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RO	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	WC	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	WC	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RO	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RO	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	WC	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	WC	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RO	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RO	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	WC	0x0
6	GPIO_QSPI_SS_EDGE_LOW	WC	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RO	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RO	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	WC	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	WC	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RO	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RO	0x0

IO_QSPI: PROC0_INTE Register

Offset: 0x34

Description

Interrupt Enable for proc0

Table 330.
PROC0_INTE Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0

位	描述	类型	复位值
22	GPIO_QSPI_SD3_EDGE_LOW	WC	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	WC	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	WC	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	WC	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	WC	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	WC	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	WC	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	WC	0x0
6	GPIO_QSPI_SS_EDGE_LOW	WC	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	WC	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	WC	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: PROC0_INTE 寄存器

偏移: 0x34

描述

proc0 的中断使能

表330。
PROC0_INTE 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: PROCO_INTF Register

Offset: 0x38

Description

Interrupt Force for proc0

Table 331.
PROCO_INTF Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: PROCO_INTF 寄存器

偏移: 0x38

描述

proc0 的中断强制

表331。
PROCO_INTF 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: PROC0_INTS Register

Offset: 0x3c

Description

Interrupt status after masking & forcing for proc0

Table 332.
PROC0_INTS Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RO	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RO	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RO	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RO	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RO	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RO	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RO	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RO	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RO	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RO	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RO	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RO	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RO	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RO	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RO	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC0_INTS 寄存器

偏移：0x3c

描述

proc0 的中断状态（掩码及强制后）

表 332。
PROC0_INTS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
7	GPIO_QSPI_SS_EDGE_HIGH	RO	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RO	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RO	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RO	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RO	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RO	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RO	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RO	0x0

IO_QSPI: PROC1_INTE Register

Offset: 0x40

Description

Interrupt Enable for proc1

Table 333.
PROC1_INTE Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0

位	描述	类型	复位值
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: PROC1_INTE 寄存器

偏移: 0x40

描述

proc1 的中断使能

表 333。
PROC1_INTE 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0

Bits	Description	Type	Reset
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: PROC1_INTF Register

Offset: 0x44

Description

Interrupt Force for proc1

Table 334.
PROC1_INTF Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: PROC1_INTS Register

Offset: 0x48

位	描述	类型	复位值
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC1_INTF 寄存器

偏移: 0x44

描述

proc1 的中断强制

表 334。
PROC1_INTF 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI：PROC1_INTS 寄存器

偏移量: 0x48

Description

Interrupt status after masking & forcing for proc1

Table 335.
PROC1_INTS Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RO	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RO	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RO	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RO	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RO	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RO	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RO	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RO	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RO	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RO	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RO	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RO	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RO	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RO	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RO	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RO	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RO	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RO	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RO	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RO	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RO	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RO	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RO	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RO	0x0

IO_QSPI: DORMANT_WAKE_INTE Register

Offset: 0x4c

Description

Interrupt Enable for dormant_wake

Table 336.
DORMANT_WAKE_INTE Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0

描述

proc1 的中断状态（掩码及强制后）

表 335。
PROC1_INTS 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

IO_QSPI: DORMANT_WAKE_INTE 寄存器

偏移量：0x4c

说明

dormant_wake 的中断使能

表 336。
DORMANT_WAKE_INTE
E 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0

Bits	Description	Type	Reset
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: DORMANT_WAKE_INTF Register

Offset: 0x50

Description

Interrupt Force for dormant_wake

Table 337.
DORMANT_WAKE_INT
F Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RW	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RW	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RW	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RW	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RW	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RW	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RW	0x0

位	描述	类型	复位值
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: DORMANT_WAKE_INTF 寄存器

偏移: 0x50

说明

dormant_wake 的中断强制

表 337。
DORMANT_WAKE_INT
F 寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	读写	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	读写	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	读写	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	读写	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	读写	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	读写	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	读写	0x0

Bits	Description	Type	Reset
16	GPIO_QSPI_SD2_LEVEL_LOW	RW	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RW	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RW	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RW	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RW	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	RW	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	RW	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	RW	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	RW	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RW	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RW	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RW	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RW	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RW	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RW	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RW	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RW	0x0

IO_QSPI: DORMANT_WAKE_INTS Register

Offset: 0x54

Description

Interrupt status after masking & forcing for dormant_wake

Table 338.
DORMANT_WAKE_INT
S Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	RO	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	RO	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	RO	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	RO	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	RO	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	RO	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	RO	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	RO	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	RO	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	RO	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	RO	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	RO	0x0

位	描述	类型	复位值
16	GPIO_QSPI_SD2_LEVEL_LOW	读写	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	读写	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	读写	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	读写	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	读写	0x0
11	GPIO_QSPI_SD0_EDGE_HIGH	读写	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	读写	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	读写	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	读写	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	读写	0x0
6	GPIO_QSPI_SS_EDGE_LOW	读写	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	读写	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	读写	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	读写	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	读写	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	读写	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	读写	0x0

IO_QSPI: DORMANT_WAKE_INTS 寄存器

偏移: 0x54

说明

dormant_wake 遮蔽及强制后的中断状态

表338。
DORMANT_WAKE_INT
S寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23	GPIO_QSPI_SD3_EDGE_HIGH	只读	0x0
22	GPIO_QSPI_SD3_EDGE_LOW	只读	0x0
21	GPIO_QSPI_SD3_LEVEL_HIGH	只读	0x0
20	GPIO_QSPI_SD3_LEVEL_LOW	只读	0x0
19	GPIO_QSPI_SD2_EDGE_HIGH	只读	0x0
18	GPIO_QSPI_SD2_EDGE_LOW	只读	0x0
17	GPIO_QSPI_SD2_LEVEL_HIGH	只读	0x0
16	GPIO_QSPI_SD2_LEVEL_LOW	只读	0x0
15	GPIO_QSPI_SD1_EDGE_HIGH	只读	0x0
14	GPIO_QSPI_SD1_EDGE_LOW	只读	0x0
13	GPIO_QSPI_SD1_LEVEL_HIGH	只读	0x0
12	GPIO_QSPI_SD1_LEVEL_LOW	只读	0x0

Bits	Description	Type	Reset
11	GPIO_QSPI_SDO_EDGE_HIGH	RO	0x0
10	GPIO_QSPI_SDO_EDGE_LOW	RO	0x0
9	GPIO_QSPI_SDO_LEVEL_HIGH	RO	0x0
8	GPIO_QSPI_SDO_LEVEL_LOW	RO	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	RO	0x0
6	GPIO_QSPI_SS_EDGE_LOW	RO	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	RO	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	RO	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	RO	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	RO	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	RO	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	RO	0x0

2.19.6.3. Pad Control - User Bank

The User Bank Pad Control registers start at a base address of `0x4001c000` (defined as `PADS_BANK0_BASE` in SDK).

Table 339. List of PADS_BANK0 registers

Offset	Name	Info
0x00	VOLTAGE_SELECT	Voltage select. Per bank control
0x04	GPIO0	Pad control register
0x08	GPIO1	Pad control register
0x0c	GPIO2	Pad control register
0x10	GPIO3	Pad control register
0x14	GPIO4	Pad control register
0x18	GPIO5	Pad control register
0x1c	GPIO6	Pad control register
0x20	GPIO7	Pad control register
0x24	GPIO8	Pad control register
0x28	GPIO9	Pad control register
0x2c	GPIO10	Pad control register
0x30	GPIO11	Pad control register
0x34	GPIO12	Pad control register
0x38	GPIO13	Pad control register
0x3c	GPIO14	Pad control register
0x40	GPIO15	Pad control register
0x44	GPIO16	Pad control register
0x48	GPIO17	Pad control register

位	描述	类型	复位值
11	GPIO_QSPI_SD0_EDGE_HIGH	只读	0x0
10	GPIO_QSPI_SD0_EDGE_LOW	只读	0x0
9	GPIO_QSPI_SD0_LEVEL_HIGH	只读	0x0
8	GPIO_QSPI_SD0_LEVEL_LOW	只读	0x0
7	GPIO_QSPI_SS_EDGE_HIGH	只读	0x0
6	GPIO_QSPI_SS_EDGE_LOW	只读	0x0
5	GPIO_QSPI_SS_LEVEL_HIGH	只读	0x0
4	GPIO_QSPI_SS_LEVEL_LOW	只读	0x0
3	GPIO_QSPI_SCLK_EDGE_HIGH	只读	0x0
2	GPIO_QSPI_SCLK_EDGE_LOW	只读	0x0
1	GPIO_QSPI_SCLK_LEVEL_HIGH	只读	0x0
0	GPIO_QSPI_SCLK_LEVEL_LOW	只读	0x0

2.19.6.3. 用户组 Pad 控制

用户组 Pad 控制寄存器的基地址起始于 `0x4001c000`（在 SDK 中定义为 `PADS_BANK0_BASE`）。

表339。 `PADS_BANK0` 寄存器列表

偏移量	名称	说明
0x00	VOLTAGE_SELECT	电压选择。按组控制
0x04	GPIO0	Pad 控制寄存器
0x08	GPIO1	Pad 控制寄存器
0x0c	GPIO2	Pad 控制寄存器
0x10	GPIO3	Pad 控制寄存器
0x14	GPIO4	Pad 控制寄存器
0x18	GPIO5	Pad 控制寄存器
0x1c	GPIO6	Pad 控制寄存器
0x20	GPIO7	Pad 控制寄存器
0x24	GPIO8	Pad 控制寄存器
0x28	GPIO9	Pad 控制寄存器
0x2c	GPIO10	Pad 控制寄存器
0x30	GPIO11	Pad 控制寄存器
0x34	GPIO12	Pad 控制寄存器
0x38	GPIO13	Pad 控制寄存器
0x3c	GPIO14	Pad 控制寄存器
0x40	GPIO15	Pad 控制寄存器
0x44	GPIO16	Pad 控制寄存器
0x48	GPIO17	Pad 控制寄存器

Offset	Name	Info
0x4c	GPIO18	Pad control register
0x50	GPIO19	Pad control register
0x54	GPIO20	Pad control register
0x58	GPIO21	Pad control register
0x5c	GPIO22	Pad control register
0x60	GPIO23	Pad control register
0x64	GPIO24	Pad control register
0x68	GPIO25	Pad control register
0x6c	GPIO26	Pad control register
0x70	GPIO27	Pad control register
0x74	GPIO28	Pad control register
0x78	GPIO29	Pad control register
0x7c	SWCLK	Pad control register
0x80	SWD	Pad control register

PADS_BANK0: VOLTAGE_SELECT Register

Offset: 0x00

Table 340.
VOLTAGE_SELECT
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Voltage select. Per bank control	RW	0x0
	Enumerated values:		
	0x0 → 3V3: Set voltage to 3.3V (DVDD >= 2V5)		
	0x1 → 1V8: Set voltage to 1.8V (DVDD ≤ 1V8)		

PADS_BANK0: GPIO0, GPIO1, ..., GPIO28, GPIO29 Registers

Offsets: 0x04, 0x08, ..., 0x74, 0x78

Description

Pad control register

Table 341. GPIO0,
GPIO1, ..., GPIO28,
GPIO29 Registers

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD : Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE : Input enable	RW	0x1
5:4	DRIVE : Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		
	0x1 → 4MA		

偏移量	名称	说明
0x4c	GPIO18	Pad 控制寄存器
0x50	GPIO19	Pad 控制寄存器
0x54	GPIO20	Pad 控制寄存器
0x58	GPIO21	Pad 控制寄存器
0x5c	GPIO22	Pad 控制寄存器
0x60	GPIO23	Pad 控制寄存器
0x64	GPIO24	Pad 控制寄存器
0x68	GPIO25	Pad 控制寄存器
0x6c	GPIO26	Pad 控制寄存器
0x70	GPIO27	Pad 控制寄存器
0x74	GPIO28	Pad 控制寄存器
0x78	GPIO29	Pad 控制寄存器
0x7c	SWCLK	Pad 控制寄存器
0x80	SWD	Pad 控制寄存器

PADS_BANK0: VOLTAGE_SELECT 寄存器

偏移: 0x00

表340。
VOLTAGE_SELECT
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	电压选择。按组控制	读写	0x0
	枚举值：		
	0x0 → 3V3：设置电压为3.3V (DVDD ≥ 2V5)		
	0x1 → 1V8：设置电压为1.8V (DVDD ≤ 1V8)		

PADS_BANK0: GPIO0、GPIO1、...、GPIO28、GPIO29 寄存器

偏移量: 0x04、0x08、...、0x74、0x78

描述

Pad 控制寄存器

表341。GPIO0,
GPIO1,...,GPIO28,
GPIO29 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD ：输出禁用。优先于外设的输出使能	读写	0x0
6	IE ：输入使能。	读写	0x1
5:4	DRIVE ：驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		

Bits	Description	Type	Reset
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE: Pull up enable	RW	0x0
2	PDE: Pull down enable	RW	0x1
1	SCHMITT: Enable schmitt trigger	RW	0x1
0	SLEWFAST: Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

PADS_BANK0: SWCLK Register

Offset: 0x7c

Description

Pad control register

Table 342. SWCLK Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD: Output disable. Has priority over output enable from peripherals	RW	0x1
6	IE: Input enable	RW	0x1
5:4	DRIVE: Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE: Pull up enable	RW	0x1
2	PDE: Pull down enable	RW	0x0
1	SCHMITT: Enable schmitt trigger	RW	0x1
0	SLEWFAST: Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

PADS_BANK0: SWD Register

Offset: 0x80

Description

Pad control register

Table 343. SWD Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD: Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE: Input enable	RW	0x1
5:4	DRIVE: Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		

位	描述	类型	复位值
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x1
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_BANK0: SWCLK 寄存器

偏移: 0x7c

描述

Pad 控制寄存器

表 342. SWCLK 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x1
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值:		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x1
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_BANK0: SWD 寄存器

偏移: 0x80

说明

引脚控制寄存器

表 343. SWD 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值:		
	0x0 → 2mA		

Bits	Description	Type	Reset
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE : Pull up enable	RW	0x1
2	PDE : Pull down enable	RW	0x0
1	SCHMITT : Enable schmitt trigger	RW	0x1
0	SLEWFAST : Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

2.19.6.4. Pad Control - QSPI Bank

The QSPI Bank Pad Control registers start at a base address of [0x40020000](#) (defined as [PADS_QSPI_BASE](#) in SDK).

Table 344. List of
PADS_QSPI registers

Offset	Name	Info
0x00	VOLTAGE_SELECT	Voltage select. Per bank control
0x04	GPIO_QSPI_SCLK	Pad control register
0x08	GPIO_QSPI_SD0	Pad control register
0x0c	GPIO_QSPI_SD1	Pad control register
0x10	GPIO_QSPI_SD2	Pad control register
0x14	GPIO_QSPI_SD3	Pad control register
0x18	GPIO_QSPI_SS	Pad control register

[PADS_QSPI](#): VOLTAGE_SELECT Register

Offset: 0x00

Table 345.
[VOLTAGE_SELECT](#)
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Voltage select. Per bank control	RW	0x0
	Enumerated values:		
	0x0 → 3V3: Set voltage to 3.3V (DVDD \geq 2V5)		
	0x1 → 1V8: Set voltage to 1.8V (DVDD \leq 1V8)		

[PADS_QSPI](#): GPIO_QSPI_SCLK Register

Offset: 0x04

Description

Pad control register

Table 346.
[GPIO_QSPI_SCLK](#)
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD : Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE : Input enable	RW	0x1

位	描述	类型	复位值
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE : 上拉使能	读写	0x1
2	PDE : 下拉使能	读写	0x0
1	SCHMITT : 施密特触发器使能	读写	0x1
0	SLEWFAST : 转换速率控制。1 = 快, 0 = 慢	读写	0x0

2.19.6.4. 引脚控制 - QSPI 组

QSPI 组引脚控制寄存器的起始基地址为 **0x40020000** (在 SDK 中定义为 PADS_QSPI_BASE)。

表 344. PADS_QSPI 寄存器列表

偏移量	名称	说明
0x00	VOLTAGE_SELECT	电压选择。按组控制
0x04	GPIO_QSPI_SCLK	Pad 控制寄存器
0x08	GPIO_QSPI_SD0	Pad 控制寄存器
0x0c	GPIO_QSPI_SD1	Pad 控制寄存器
0x10	GPIO_QSPI_SD2	Pad 控制寄存器
0x14	GPIO_QSPI_SD3	Pad 控制寄存器
0x18	GPIO_QSPI_SS	Pad 控制寄存器

PADS_QSPI: VOLTAGE_SELECT 寄存器

偏移: 0x00

表 345. VOLTAGE_SELECT 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	电压选择。按组控制 枚举值: 0x0 → 3V3: 设置电压为3.3V (DVDD ≥ 2V5) 0x1 → 1V8: 设置电压为1.8V (DVDD ≤ 1V8)	读写	0x0

PADS_QSPI: GPIO_QSPI_SCLK 寄存器

偏移量: 0x04

描述

引脚控制寄存器

表 346. GPIO_QSPI_SCLK 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD : 输出禁用。优先于外设的输出使能	读写	0x0
6	IE : 输入使能。	读写	0x1

Bits	Description	Type	Reset
5:4	DRIVE: Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE: Pull up enable	RW	0x0
2	PDE: Pull down enable	RW	0x1
1	SCHMITT: Enable schmitt trigger	RW	0x1
0	SLEWFAST: Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

PADS_QSPI: GPIO_QSPI_SD0, GPIO_QSPI_SD1, GPIO_QSPI_SD2, GPIO_QSPI_SD3 Registers

Offsets: 0x08, 0x0c, 0x10, 0x14

Description

Pad control register

Table 347.
GPIO_QSPI_SD0,
GPIO_QSPI_SD1,
GPIO_QSPI_SD2,
GPIO_QSPI_SD3
Registers

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD: Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE: Input enable	RW	0x1
5:4	DRIVE: Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE: Pull up enable	RW	0x0
2	PDE: Pull down enable	RW	0x0
1	SCHMITT: Enable schmitt trigger	RW	0x1
0	SLEWFAST: Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

PADS_QSPI: GPIO_QSPI_SS Register

Offset: 0x18

Description

Pad control register

位	描述	类型	复位值
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x1
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_QSPI: GPIO_QSPI_SD0, GPIO_QSPI_SD1, GPIO_QSPI_SD2, GPIO_QS PI_SD3 寄存器

偏移量：0x08, 0x0c, 0x10, 0x14

描述

Pad 控制寄存器

表 347。
GPIO_QSPI_SD0
、GPIO_QSPI_SD1
、GPIO_QSPI_SD2
、GPIO_QSPI_SD3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x0
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

PADS_QSPI: GPIO_QSPI_SS 寄存器

偏移量：0x18

描述

引脚控制寄存器

Table 348.
GPIO_QSPI_SS
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	OD: Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE: Input enable	RW	0x1
5:4	DRIVE: Drive strength.	RW	0x1
	Enumerated values:		
	0x0 → 2MA		
	0x1 → 4MA		
	0x2 → 8MA		
	0x3 → 12MA		
3	PUE: Pull up enable	RW	0x1
2	PDE: Pull down enable	RW	0x0
1	SCHMITT: Enable schmitt trigger	RW	0x1
0	SLEWFAST: Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

2.20. Sysinfo

2.20.1. Overview

The sysinfo block contains system information. The first register contains the Chip ID, which allows the programmer to know which version of the chip software is running on. The second register will always read as 1 on the device.

2.20.2. List of Registers

The sysinfo registers start at a base address of `0x40000000` (defined as `SYSINFO_BASE` in SDK).

Table 349. List of
SYSINFO registers

Offset	Name	Info
0x00	<code>CHIP_ID</code>	JEDEC JEP-106 compliant chip identifier.
0x04	<code>PLATFORM</code>	Platform register. Allows software to know what environment it is running in.
0x40	<code>GITREF_RP2040</code>	Git hash of the chip source. Used to identify chip version.

SYSINFO: CHIP_ID Register

Offset: 0x00

Description

JEDEC JEP-106 compliant chip identifier.

Table 350. CHIP_ID
Register

Bits	Description	Type	Reset
31:28	REVISION	RO	-
27:12	PART	RO	-

表 348。
GPIO_QSPI_SS
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	OD: 输出禁用。优先于外设的输出使能	读写	0x0
6	IE: 输入使能。	读写	0x1
5:4	DRIVE: 驱动强度。	读写	0x1
	枚举值：		
	0x0 → 2mA		
	0x1 → 4mA		
	0x2 → 8mA		
	0x3 → 12mA		
3	PUE: 上拉使能	读写	0x1
2	PDE: 下拉使能	读写	0x0
1	SCHMITT: 施密特触发器使能	读写	0x1
0	SLEWFAST: 转换速率控制。1 = 快, 0 = 慢	读写	0x0

2.20. 系统信息

2.20.1. 概述

sysinfo 区块包含系统信息。第一个寄存器包含芯片 ID，允许程序员识别软件运行的芯片版本。设备上的第二个寄存器始终读取为 1。

2.20.2. 寄存器列表

sysinfo 寄存器起始基址为 [0x40000000](#)(定义于SDK中的SYSINFO_BASE)。

表349。SYSINF
0 寄存器列表

偏移量	名称	说明
0x00	CHIP_ID	符合JEDEC JEP-106标准的芯片标识符。
0x04	平台	平台寄存器。允许软件识别其运行环境。
0x40	GITREF_RP2040	芯片源代码的Git哈希值。用于识别芯片版本。

SYSINFO：CHIP_ID 寄存器

偏移: 0x00

描述

符合JEDEC JEP-106标准的芯片标识符。

表350。CHIP_ID
寄存器

位	描述	类型	复位值
31:28	修订版本	只读	-
27:12	部件	只读	-

Bits	Description	Type	Reset
11:0	MANUFACTURER	RO	-

SYSINFO: PLATFORM Register

Offset: 0x04

Description

Platform register. Allows software to know what environment it is running in.

Table 351. PLATFORM Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	ASIC	RO	0x0
0	FPGA	RO	0x0

SYSINFO: GITREF_RP2040 Register

Offset: 0x40

Table 352. GITREF_RP2040 Register

Bits	Description	Type	Reset
31:0	Git hash of the chip source. Used to identify chip version.	RO	-

2.21. Syscfg

2.21.1. Overview

The system config block controls miscellaneous chip settings including:

- NMI (Non-Maskable-Interrupt) mask to pick sources that generate the NMI
- Processor config
 - DAP Instance ID (to change the address that the SWD uses to communicate with the core in debug)
 - Processor status (If the processor is halted, which may be useful in debug)
- Processor IO config
 - Input synchroniser control (to allow input synchronisers to be bypassed to reduce latency where clocks are synchronous)
- Debug control
 - Provides the ability to control the SWD interface from inside the chip. This means Core 0 could debug Core 1, which may make debug connectivity easier.
- Memory power down (each memory can be powered down if not being used to save a small amount of extra power).

2.21.2. List of Registers

The system config registers start at a base address of **0x40004000** (defined as **SYSCFG_BASE** in SDK).

位	描述	类型	复位值
11:0	制造商	只读	-

SYSINFO：PLATFORM寄存器

偏移量: 0x04

说明

平台寄存器。允许软件识别其运行环境。

表351。PLATFORM
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	ASIC	只读	0x0
0	FPGA	只读	0x0

SYSINFO: GITREF_RP2040 寄存器

偏移: 0x40

表 352。
GITREF_RP2040
寄存器

位	描述	类型	复位值
31:0	芯片源代码的Git哈希值。用于识别芯片版本。	只读	-

2.21. Syscfg

2.21.1. 概述

系统配置块控制包括以下杂项芯片设置：

- NMI（不可屏蔽中断）屏蔽，用以选择产生 NMI 的源
- 处理器配置
 - DAP 实例 ID（用于更改 SWD 与核心调试通信时所用的地址）
 - 处理器状态（处理器是否停止，此信息在调试时可能有用）
- 处理器 IO 配置
 - 输入同步器控制（允许绕过输入同步器以减少时钟同步情况下的延迟）
 - 调试控制
 - 提供从芯片内部控制 SWD 接口的能力。这意味着核心0能够调试核心1，可能使调试连接更为便捷。
- 存储器断电（未使用时，单个存储器可断电以节省少量额外功耗）。

2.21.2. 寄存器列表

系统配置寄存器起始基址为 **0x40004000**（在 SDK 中定义为 SYSCFG_BASE）。

Table 353. List of SYSCFG registers

Offset	Name	Info
0x00	PROC0_NMI_MASK	Processor core 0 NMI source mask
0x04	PROC1_NMI_MASK	Processor core 1 NMI source mask
0x08	PROC_CONFIG	Configuration for processors
0x0c	PROC_IN_SYNC_BYPASS	For each bit, if 1, bypass the input synchronizer between that GPIO and the GPIO input register in the SIO. The input synchronizers should generally be unbypassed, to avoid injecting metastabilities into processors. If you're feeling brave, you can bypass to save two cycles of input latency. This register applies to GPIO 0...29.
0x10	PROC_IN_SYNC_BYPASS_HI	For each bit, if 1, bypass the input synchronizer between that GPIO and the GPIO input register in the SIO. The input synchronizers should generally be unbypassed, to avoid injecting metastabilities into processors. If you're feeling brave, you can bypass to save two cycles of input latency. This register applies to GPIO 30...35 (the QSPI IOs).
0x14	DBGFORCE	Directly control the SWD debug port of either processor
0x18	MEMPOWERDOWN	Control power downs to memories. Set high to power down memories. Use with extreme caution

SYSCFG: PROC0_NMI_MASK Register

Offset: 0x00

Description

Processor core 0 NMI source mask

Table 354.
PROC0_NMI_MASK
Register

Bits	Description	Type	Reset
31:0	Set a bit high to enable NMI from that IRQ	RW	0x00000000

SYSCFG: PROC1_NMI_MASK Register

Offset: 0x04

Description

Processor core 1 NMI source mask

Table 355.
PROC1_NMI_MASK
Register

Bits	Description	Type	Reset
31:0	Set a bit high to enable NMI from that IRQ	RW	0x00000000

SYSCFG: PROC_CONFIG Register

Offset: 0x08

Description

Configuration for processors

表 353. SYSCFG 寄存器列表

偏移量	名称	说明
0x00	PROC0_NMI_MASK	处理器核 0 NMI 源掩码
0x04	PROC1_NMI_MASK	处理器核 1 NMI 源掩码
0x08	PROC_CONFIG	处理器配置
0x0c	PROC_IN_SYNC_BYPASS	对于每个位，若为 1，则绕过该 GPIO 与 SIO 中 GPIO 输入寄存器之间的输入同步器。输入同步器通常应 保持不绕过，以避免向处理器注入亚稳态。 若您有足够的把握，可绕过同步器以节省两个输入 延迟周期。该寄存器适用于 GPIO 0 至 29。
0x10	PROC_IN_SYNC_BYPASS_HI	对于每个位，若为 1，则绕过该 GPIO 与 SIO 中 GPIO 输入寄存器之间的输入同步器。输入同步器通常应 保持不绕过，以避免向处理器注入亚稳态。 若您有足够的把握，可绕过同步器以节省两个输入 延迟周期。该寄存器适用于 GPIO 30 至 35 (QSPI IO)。
0x14	DBGFORCE	直接控制任一处理器的 SWD 调试端口
0x18	MEMPOWERDOWN	控制内存的掉电状态。置高电平以关闭存储器电源。 请务必谨慎使用

SYSCFG: PROC0_NMI_MASK 寄存器

偏移: 0x00

描述

处理器核 0 NMI 源掩码

表 354.
PROC0_NMI_MASK
寄存器

位	描述	类型	复位值
31:0	将对应位设置为高以启用来自该 IRQ 的 NMI	读写	0x00000000

SYSCFG: PROC1_NMI_MASK 寄存器

偏移量: 0x04

说明

处理器核 1 NMI 源掩码

表 355.
PROC1_NMI_MASK
寄存器

位	描述	类型	复位值
31:0	将对应位设置为高以启用来自该 IRQ 的 NMI	读写	0x00000000

SYSCFG: PROC_CONFIG 寄存器

偏移量: 0x08

说明

处理器配置

Table 356.
PROC_CONFIG
Register

Bits	Description	Type	Reset
31:28	PROC1_DAP_INSTID : Configure proc1 DAP instance ID. Recommend that this is NOT changed until you require debug access in multi-chip environment WARNING: do not set to 15 as this is reserved for RescueDP	RW	0x1
27:24	PROC0_DAP_INSTID : Configure proc0 DAP instance ID. Recommend that this is NOT changed until you require debug access in multi-chip environment WARNING: do not set to 15 as this is reserved for RescueDP	RW	0x0
23:2	Reserved.	-	-
1	PROC1_HALTED : Indication that proc1 has halted	RO	0x0
0	PROC0_HALTED : Indication that proc0 has halted	RO	0x0

SYSCFG: PROC_IN_SYNC_BYPASS Register

Offset: 0x0c

Table 357.
PROC_IN_SYNC_BYPA
SS Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	For each bit, if 1, bypass the input synchronizer between that GPIO and the GPIO input register in the SIO. The input synchronizers should generally be unbypassed, to avoid injecting metastabilities into processors. If you're feeling brave, you can bypass to save two cycles of input latency. This register applies to GPIO 0...29.	RW	0x00000000

SYSCFG: PROC_IN_SYNC_BYPASS_HI Register

Offset: 0x10

Table 358.
PROC_IN_SYNC_BYPA
SS_HI Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	For each bit, if 1, bypass the input synchronizer between that GPIO and the GPIO input register in the SIO. The input synchronizers should generally be unbypassed, to avoid injecting metastabilities into processors. If you're feeling brave, you can bypass to save two cycles of input latency. This register applies to GPIO 30...35 (the QSPI IOs).	RW	0x00

SYSCFG: DBGFORCE Register

Offset: 0x14

Description

Directly control the SWD debug port of either processor

Table 359. DBGFORCE
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	PROC1_ATTACH : Attach processor 1 debug port to syscfg controls, and disconnect it from external SWD pads.	RW	0x0
6	PROC1_SWCLK : Directly drive processor 1 SWCLK, if PROC1_ATTACH is set	RW	0x1

表 356。
PROC_CONFIG
寄存器

位	描述	类型	复位值
31:28	PROC1_DAP_INSTID : 配置 proc1 DAP 实例 ID。 建议仅在多芯片环境中需要调试访问时更改此项 警告：请勿设置为 15，该值保留供 RescueDP 使用	读写	0x1
27:24	PROC0_DAP_INSTID : 配置 proc0 DAP 实例 ID。 建议仅在多芯片环境中需要调试访问时更改此项 警告：请勿设置为 15，该值保留供 RescueDP 使用	读写	0x0
23:2	保留。	-	-
1	PROC1_HALTED : 指示 proc1 已停止	只读	0x0
0	PROC0_HALTED : 指示 proc0 已停止	只读	0x0

SYSCFG: PROC_IN_SYNC_BYPASS 寄存器

偏移: 0x0c

表 357。
PROC_IN_SYNC_BYPA
SS 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29:0	对于每个位，如果为1，则绕过SIO中该GPIO与GPIO输入寄存器之间的输入同步器。一般应避免绕过输入同步器，以防将亚稳态注入处理器。 如果您有胆量，可以通过绕过来节省两个周期的输入延迟。该寄存器适用于 GPIO 0 至 29。	读写	0x00000000

SYSCFG: PROC_IN_SYNC_BYPASS_HI 寄存器

偏移: 0x10

表 358。
PROC_IN_SYNC_BYP
ASS_HI 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	对于每个位，如果为1，则绕过SIO中该GPIO与GPIO输入寄存器之间的输入同步器。一般应避免绕过输入同步器，以防将亚稳态注入处理器。 如果您有胆量，可以通过绕过来节省两个周期的输入延迟。该寄存器适用于 GPIO 30 至 35 (QSPI IO)。	读写	0x00

SYSCFG: DBGFORCE 寄存器

偏移: 0x14

说明

直接控制任一处理器的 SWD 调试端口

表 359: DBGFORCE
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	PROC1_ATTACH : 将处理器1调试端口连接至syscfg控制，并断开其与外部SWD引脚的连接。	读写	0x0
6	PROC1_SWCLK : 若设置了PROC1_ATTACH，则直接驱动处理器1的SWCLK信号。	读写	0x1

Bits	Description	Type	Reset
5	PROC1_SWDI: Directly drive processor 1 SWDIO input, if PROC1_ATTACH is set	RW	0x1
4	PROC1_SWDO: Observe the value of processor 1 SWDIO output.	RO	-
3	PROC0_ATTACH: Attach processor 0 debug port to syscfg controls, and disconnect it from external SWD pads.	RW	0x0
2	PROC0_SWCLK: Directly drive processor 0 SWCLK, if PROC0_ATTACH is set	RW	0x1
1	PROC0_SWDI: Directly drive processor 0 SWDIO input, if PROC0_ATTACH is set	RW	0x1
0	PROC0_SWDO: Observe the value of processor 0 SWDIO output.	RO	-

SYSCFG: MEMPOWERDOWN Register

Offset: 0x18

Description

Control power downs to memories. Set high to power down memories.

Use with extreme caution

Table 360.
MEMPOWERDOWN
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	ROM	RW	0x0
6	USB	RW	0x0
5	SRAM5	RW	0x0
4	SRAM4	RW	0x0
3	SRAM3	RW	0x0
2	SRAM2	RW	0x0
1	SRAM1	RW	0x0
0	SRAM0	RW	0x0

2.22. TBMAN

TBMAN refers to the testbench manager, which is used during chip development simulations to verify the design. During these simulations TBMAN allows software running on RP2040 to control the testbench and simulation environment. On the real chip it has no effect other than providing a single **PLATFORM** register to indicate that this is the real chip. This **PLATFORM** functionality is duplicated in the sysinfo ([Section 2.20](#)) registers.

2.22.1. List of Registers

The TBMAN registers start at a base address of **0x4006c000** (defined as **TBMAN_BASE** in SDK).

位	描述	类型	复位值
5	PROC1_SWDI : 若设置了PROC1_ATTACH，则直接驱动处理器1的SWDIO输入信号。	读写	0x1
4	PROC1_SWDO : 观察处理器1的SWDIO输出信号值。	只读	-
3	PROC0_ATTACH : 将处理器0的调试端口连接至syscfg控制单元，同时断开其与外部SWD焊盘的连接。	读写	0x0
2	PROC0_SWCLK : 若设置了PROC0_ATTACH，则直接驱动处理器0的SWCLK信号。	读写	0x1
1	PROC0_SWDI : 若设置了PROC0_ATTACH，则直接驱动处理器0的SWDIO输入信号。	读写	0x1
0	PROC0_SWDO : 监测处理器0的SWDIO输出值。	只读	-

SYSCFG：MEMPOWERDOWN寄存器

偏移：0x18

说明

控制内存的掉电状态。置高以断电存储器。

请务必谨慎使用

表360。
MEMPOWERDOWN
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	ROM	读写	0x0
6	USB	读写	0x0
5	SRAM5	读写	0x0
4	SRAM4	读写	0x0
3	SRAM3	读写	0x0
2	SRAM2	读写	0x0
1	SRAM1	读写	0x0
0	SRAM0	读写	0x0

2.22. TBMAN

TBMAN指测试台管理器，用于芯片开发仿真中验证设计的功能。

在这些仿真过程中，TBMAN允许运行于RP2040上的软件控制测试台及仿真环境。在真实芯片中，其唯一作用是提供一个单一的 PLATFORM 寄存器，以表明芯片为真实硬件。该 PLATFORM 功能在 sysinfo (第2.20节) 寄存器中亦有重复实现。

2.22.1. 寄存器列表

TBMAN 寄存器起始地址为 **0x4006c000** (在 SDK 中定义为 TBMAN_BASE)。

Table 361. List of TBMAN registers

Offset	Name	Info
0x0	PLATFORM	Indicates the type of platform in use

TBMAN: PLATFORM Register

Offset: 0x0

Description

Indicates the type of platform in use

Table 362. PLATFORM Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	FPGA: Indicates the platform is an FPGA	RO	0x0
0	ASIC: Indicates the platform is an ASIC	RO	0x1

表 361.
TBMAN 寄存器列表

偏移量	名称	说明
0x0	平台	指示所使用的平台类型

TBMAN：PLATFORM 寄存器

偏移: 0x0

描述

指示所使用的平台类型

表 362.
PLATFORM 寄存器

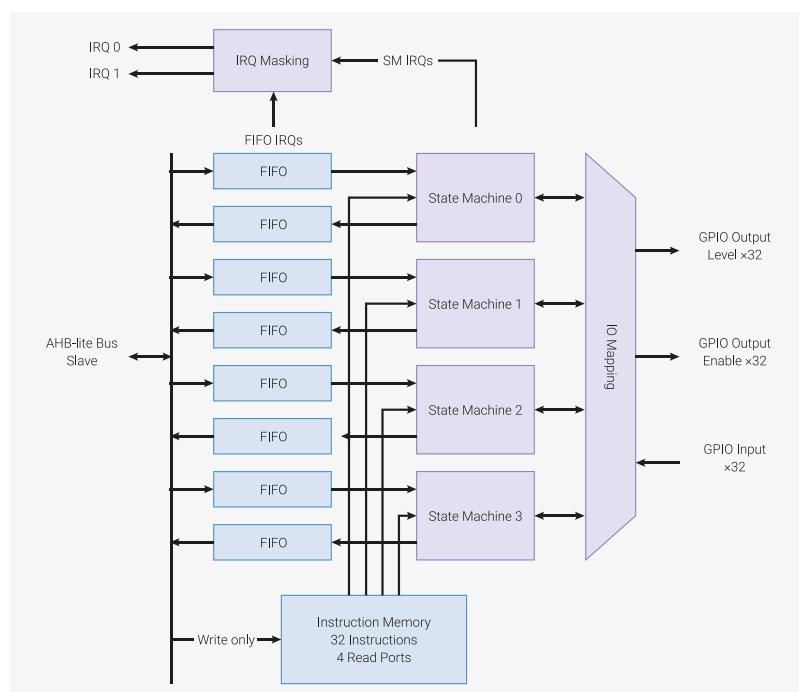
位	描述	类型	复位值
31:2	保留。	-	-
1	FPGA : 表示平台为 FPGA	只读	0x0
0	ASIC : 表示平台为 ASIC	只读	0x1

Chapter 3. PIO

3.1. Overview

There are 2 identical PIO blocks in RP2040. Each PIO block has dedicated connections to the bus fabric, GPIO and interrupt controller. The diagram for a single PIO block is show in [Figure 38](#).

Figure 38. PIO block-level diagram. There are two PIO blocks with four state machines each. The four state machines simultaneously execute programs from a shared instruction memory. FIFO data queues buffer data transferred between PIO and the system. GPIO mapping logic allows each state machine to observe and manipulate up to 30 GPIOs.



The programmable input/output block (PIO) is a versatile hardware interface. It can support a variety of IO standards, including:

- 8080 and 6800 parallel bus
- I²C
- 3-pin I²S
- SDIO
- SPI, DSPI, QSPI
- UART
- DPI or VGA (via resistor DAC)

PIO is programmable in the same sense as a processor. There are two PIO blocks with four state machines each, that can independently execute sequential programs to manipulate GPIOs and transfer data. Unlike a general purpose processor, PIO state machines are highly specialised for IO, with a focus on determinism, precise timing, and close integration with fixed-function hardware. Each state machine is equipped with:

- Two 32-bit shift registers – either direction, any shift count
- Two 32-bit scratch registers
- 4x32-bit bus FIFO in each direction (TX/RX), reconfigurable as 8x32 in a single direction
- Fractional clock divider (16 integer, 8 fractional bits)

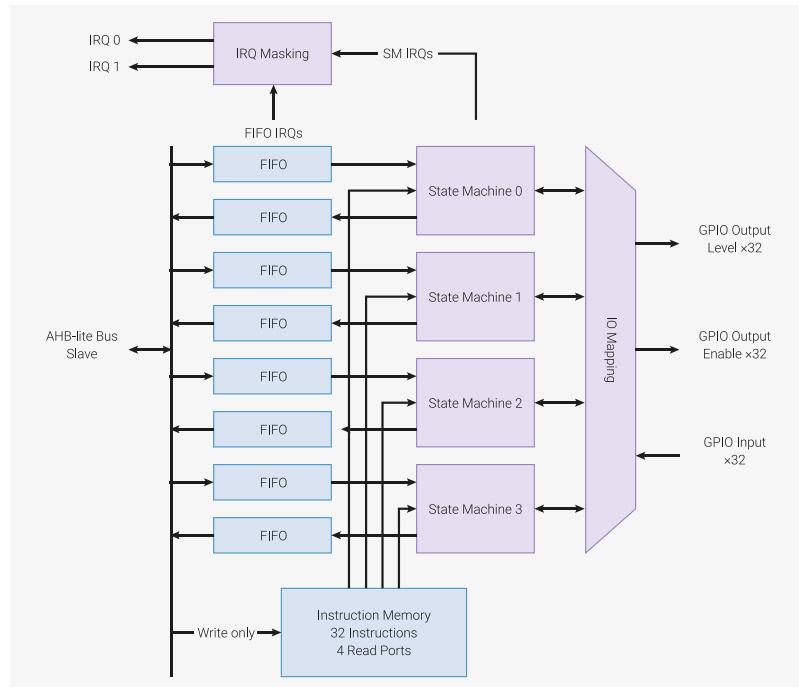
第3章. PIO

3.1. 概述

RP2040 中有两个相同的 PIO 模块。每个 PIO 模块均具备专用的总线结构、GPIO 和中断控制器连接。单个 PIO 模块的示意图如图 38 所示。

图 38。PIO 模块结构示意图。共有两个 PIO 模块，每个模块包含四个状态机。这四个状态机同时从共享的指令存储器执行程序。

FIFO 数据队列缓冲在 PIO 与系统之间传输的数据。GPIO 映射逻辑允许每个状态机监视并操作最多 30 个 G PIO。



可编程输入/输出模块（PIO）是一种多功能硬件接口。它支持多种输入输出标准，包括：

- 8080和6800并行总线
- I2C
- 3针I2S
- SDIO
- SPI、DSPI、QSPI
- UART
- DPI或VGA（通过电阻式数模转换器）

PIO 的编程方式与处理器类似。共有两个 PIO 模块，每个模块包含四个状态机，可独立执行顺序程序以控制 GPIO 及传输数据。与通用处理器不同，PIO 状态机高度专用于 IO，强调确定性、精确时序及与定功能硬件的紧密集成。每个状态机配备：

- 两个32位移位寄存器——任意方向，任意移位位数
- 两个32位临时寄存器
- 4×32位总线FIFO，双向（发送/接收），可重新配置为单向8×32
- 分数时钟分频器（16位整数，8位小数）

- Flexible GPIO mapping
- DMA interface, sustained throughput up to 1 word per clock from system DMA
- IRQ flag set/clear/status

Each state machine, along with its supporting hardware, occupies approximately the same silicon area as a standard serial interface block, such as an SPI or I2C controller. However, PIO state machines can be configured and reconfigured dynamically to implement numerous different interfaces.

Making state machines programmable in a software-like manner, rather than a fully configurable logic fabric like a CPLD, allows more hardware interfaces to be offered in the same cost and power envelope. This also presents a more familiar programming model, and simpler tool flow, to those who wish to exploit PIO's full flexibility by programming it directly, rather than using a premade interface from the PIO library.

PIO is highly performant as well as flexible, thanks to a carefully selected set of fixed-function hardware inside each state machine. When outputting DPI, PIO can sustain 360Mbps during the active scanline period when running from a 48MHz system clock. In this example, one state machine is handling frame/scanline timing and generating the pixel clock, while another is handling the pixel data, and unpacking run-length-encoded scanlines.

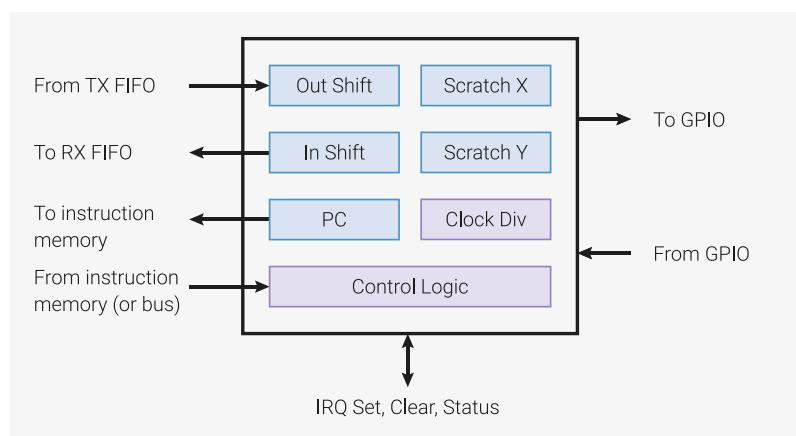
State machines' inputs and outputs are mapped to up to 32 GPIOs (limited to 30 GPIOs for RP2040), and all state machines have independent, simultaneous access to any GPIO. For example, the standard UART code allows TX, RX, CTS and RTS to be any four arbitrary GPIOs, and I2C permits the same for SDA and SCL. The amount of freedom available depends on how exactly a given PIO program chooses to use PIO's pin mapping resources, but at the minimum, an interface can be freely shifted up or down by some number of GPIOs.

3.2. Programmer's Model

The four state machines execute from a shared instruction memory. System software loads programs into this memory, configures the state machines and IO mapping, and then sets the state machines running. PIO programs come from various sources: assembled directly by the user, drawn from the PIO library, or generated programmatically by user software.

From this point on, state machines are generally autonomous, and system software interacts through DMA, interrupts and control registers, as with other peripherals on RP2040. For more complex interfaces, PIO provides a small but flexible set of primitives which allow system software to be more hands-on with state machine control flow.

Figure 39. State machine overview.
Data flows in and out through a pair of FIFOs. The state machine executes a program which transfers data between these FIFOs, a set of internal registers, and the pins. The clock divider can reduce the state machine's execution speed by a constant factor.



3.2.1. PIO Programs

PIO state machines execute short, binary programs.

Programs for common interfaces, such as UART, SPI, or I2C, are available in the PIO library, so in many cases, it is not necessary to write PIO programs. However, the PIO is much more flexible when programmed directly, supporting a wide

- 灵活的GPIO映射
- DMA接口，系统DMA可持续提供高达每时钟周期1个字的吞吐量
- IRQ标志的设置／清除／状态

每个状态机及其支持硬件占用的硅片面积大致相当于标准串行接口模块，例如SPI或I2C控制器。然而，PIO状态机可动态配置和重新配置，以实现多种不同接口。

通过将状态机设计为类似软件的可编程方式，而非像CPLD那样的完全可配置逻辑结构，可在相同成本和功耗范围内提供更多硬件接口。这同时为希望通过直接编程而非使用预置PIO库接口充分利用PIO灵活性的用户，提供了更为熟悉的编程模型和更简便的工具流程。

PIO不仅灵活且高性能，这得益于每个状态机内经过精心挑选的固定功能硬件。在以48MHz系统时钟运行并输出DPI时，PIO能在活动扫描线期间持续提供360Mbps的传输速率。在本示例中，一个状态机负责帧和扫描线时序并生成像素时钟，另一个状态机负责像素数据，并对运行长度编码的扫描线进行解码。

状态机的输入和输出可映射至最多32个GPIO（RP2040限制为30个GPIO），且所有状态机均可独立且同时访问任意GPIO。例如，标准UART代码允许TX、RX、CTS和RTS为任意四个GPIO，I2C同样允许SDA和SCL为任意GPIO。可用自由度取决于特定PIO程序如何利用PIO的引脚映射资源，但至少可以在一定范围内自由调整接口对应的GPIO编号。

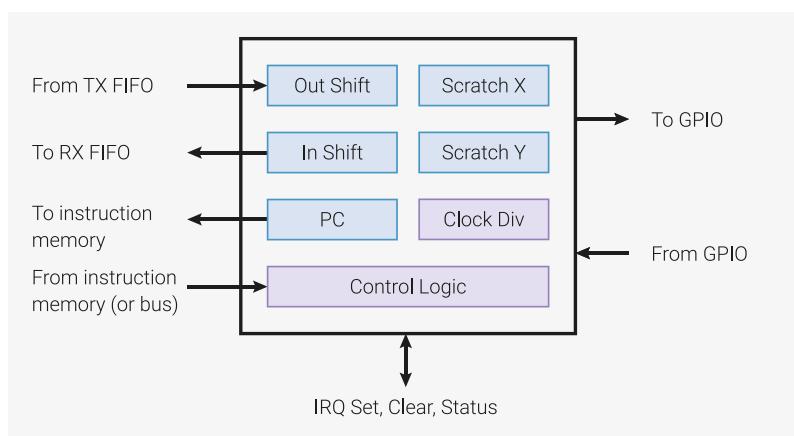
3.2. 程序员模型

四个状态机共享执行一段指令存储器。系统软件将程序加载到该存储器，配置状态机及IO映射，然后启动状态机运行。PIO程序来源多样：用户直接汇编、来自PIO库，或由用户软件编程生成。

从此点开始，状态机通常是自主运行的，系统软件通过DMA、中断和控制寄存器与之交互，如同RP2040上的其他外设。对于更复杂的接口，PIO提供了一套小巧而灵活的原语，允许系统软件更直接地控制状态机的执行流程。

图 39：状态机概述。
数据通过一对FIFO流入和流出。
状态机执行程序，在这些FIFO、一组内部寄存器与引脚之间传输数据。

时钟分频器可将状态机的执行速度按固定比例降低。



3.2.1. PIO 程序

PIO状态机执行简短的二进制程序。

常用接口如UART、SPI或I2C的程序已包含于PIO库中，因此在许多情况下无需编写新的PIO程序。然而，PIO在直接编程时具有更高的灵活性，支持更广泛的

variety of interfaces which may not have been foreseen by its designers.

The PIO has a total of nine instructions: `JMP`, `WAIT`, `IN`, `OUT`, `PUSH`, `PULL`, `MOV`, `IRQ`, and `SET`. See [Section 3.4](#) for details on these instructions.

Though the PIO only has a total of nine instructions, it would be difficult to edit PIO program binaries by hand. PIO assembly is a textual format, describing a PIO program, where each command corresponds to one instruction in the output binary. Below is an example program in PIO assembly:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> Lines 8 - 13

```
8 .program squarewave
9     set pindirs, 1    ; Set pin to output
10    again:
11        set pins, 1 [1] ; Drive pin high and then delay for one cycle
12        set pins, 0      ; Drive pin low
13        jmp again       ; Set PC to label `again`
```

The PIO assembler is included with the SDK, and is called `pioasm`. This program processes a PIO assembly input text file, which may contain multiple programs, and writes out the assembled programs ready for use. For the SDK these assembled programs are emitted in form of C headers, containing constant arrays: For more information see [Section 3.3](#)

3.2.2. Control Flow

On every system clock cycle, each state machine fetches, decodes and executes one instruction. Each instruction takes precisely one cycle, unless it explicitly stalls (such as the `WAIT` instruction). Instructions may also insert a delay of up to 31 cycles before the next instruction is executed to aid the writing of cycle-exact programs.

The program counter, or `PC`, points to the location in the instruction memory being executed on this cycle. Generally, `PC` increments by one each cycle, wrapping at the end of the instruction memory. Jump instructions are an exception and explicitly provide the next value that `PC` will take.

Our example assembly program (listed as `.program squarewave` above) shows both of these concepts in practice. It drives a 50/50 duty cycle square wave onto a GPIO, with a period of four cycles. Using some other features (e.g. side-set) this can be made as low as two cycles.

NOTE

Side-set is where a state machine drives a small number of GPIOs *in addition* to the main side effects of the instruction it executes. It's described fully in [Section 3.5.1](#).

The system has write-only access to the instruction memory, which is used to load programs:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> Lines 34 - 38

```
34     // Load the assembled program directly into the PIO's instruction memory.
35     // Each PIO instance has a 32-slot instruction memory, which all 4 state
36     // machines can see. The system has write-only access.
37     for (uint i = 0; i < count_of(squarewave_program_instructions); ++i)
38         pio->instr_mem[i] = squarewave_program_instructions[i];
```

The clock divider slows the state machine's execution by a constant factor, represented as a 16.8 fixed-point fractional number. Using the above example, if a clock division of `2.5` were programmed, the square wave would have a period of $4 \times 2.5 = 10$ cycles. This is useful for setting a precise baud rate for a serial interface, such as a UART.

接口类型，这些接口可能并未被其设计者预见。

PIO 总共有九条指令：`JMP`、`WAIT`、`IN`、`OUT`、`PUSH`、`PULL`、`MOV`、`IRQ` 和 `SET`。有关这些指令的详细信息，请参见第 3.4 节。

尽管 PIO 仅包含九条指令，但手工编辑 PIO 程序的二进制文件仍十分困难。PIO 汇编是一种文本格式，用以描述 PIO 程序，其中每条指令对应输出二进制文件中的一条机器指令。以下为 PIO 汇编示例程序：

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> 第 8 至 13 行

```
8 .program squarewave
9     set pindirs, 1      ; 设置引脚为输出
10 again:
11     set pins, 1 [1]    ; 将引脚置为高电平，随后延迟一个周期
12     set pins, 0        ; 将引脚置为低电平
13     jmp again          ; 将程序计数器设置为标签 `again`
```

PIO 汇编器包含于 SDK 中，名称为 `pioasm`。该程序处理 PIO 汇编输入文本文件，该文件可能包含多个程序，并输出已组装且可用的程序。对于 SDK，这些组装好的程序以 C 头文件形式输出，包含常量数组：详情请参见第 3.3 节。

3.2.2. 控制流

每个系统时钟周期中，每个状态机都会获取、解码并执行一条指令。每条指令严格占用一个周期，除非指令明确暂停（例如 `WAIT` 指令）。指令还可在执行下一条指令前插入最多 31 个周期的延迟，以辅助编写周期精确的程序。

程序计数器（`PC`）指向本周期正在执行的指令内存地址。通常，`PC` 每周期递增 1，指令内存末端则回绕。跳转指令为例外，它们明确指定 `PC` 的下一取值。

我们的示例汇编程序（如上所示，标记为 `.program squarewave`）展示了这两个概念的实际应用。它驱动一个 50/50 占空比的方波输出到 GPIO，周期为四个时钟周期。利用其他特性（例如 side-set），该周期可缩短至两个时钟周期。

注意

Side-set 指状态机在执行指令的主要副作用之外，同时驱动少量 GPIO 的功能。详细描述见第 3.5.1 节。

系统对指令存储器具有写权限，仅用于加载程序：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 34 行至第 38 行

```
34 // 将汇编程序直接加载至 PIO 的指令存储器。
35 // 每个 PIO 实例拥有 32 槽位指令存储器，4 个状态机均可访问。
36 // 系统仅对其具有写权限。
37 for (uint i = 0; i < count_of(squarewave_program_instructions); ++i)
38     pio->instr_mem[i] = squarewave_program_instructions[i];
```

时钟分频器通过一个常数因子减缓状态机的执行速度，该因子以 16.8 定点小数表示。以上述示例为例，若设置时钟分频为 2.5，则方波的周期将以时钟周期数计算。此功能对于设置串行接口（如 UART）的精确波特率尤为重要。

$$4 \times 2.5 = 10$$

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> Lines 42 - 47

```

42     // Configure state machine 0 to run at sysclk/2.5. The state machines can
43     // run as fast as one instruction per clock cycle, but we can scale their
44     // speed down uniformly to meet some precise frequency target, e.g. for a
45     // UART baud rate. This register has 16 integer divisor bits and 8
46     // fractional divisor bits.
47     pio->sm[0].clkdiv = (uint32_t) (2.5f * (1 << 16));

```

The above code fragments are part of a complete code example which drives a 12.5MHz square wave out of GPIO 0 (or any other pins we might choose to map). We can also use pins **WAIT PIN** instruction to stall a state machine's execution for some amount of time, or a **JMP PIN** instruction to branch on the state of a pin, so control flow can vary based on pin state.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> Lines 51 - 59

```

51     // There are five pin mapping groups (out, in, set, side-set, jmp pin)
52     // which are used by different instructions or in different circumstances.
53     // Here we're just using SET instructions. Configure state machine 0 SETs
54     // to affect GPIO 0 only; then configure GPIO0 to be controlled by PIO0,
55     // as opposed to e.g. the processors.
56     pio->sm[0].pinctrl =
57         (1 << PIO_SM0_PINCTRL_SET_COUNT_LSB) |
58         (0 << PIO_SM0_PINCTRL_SET_BASE_LSB);
59     gpio_set_function(0, pio_get_funcsel(pio));

```

The system can start and stop each state machine at any time, via the CTRL register. Multiple state machines can be started simultaneously, and the deterministic nature of PIO means they can stay perfectly synchronised.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> Lines 63 - 67

```

63     // Set the state machine running. The PIO CTRL register is global within a
64     // PIO instance, so you can start/stop multiple state machines
65     // simultaneously. We're using the register's hardware atomic set alias to
66     // make one bit high without doing a read-modify-write on the register.
67     hw_set_bits(&pio->ctrl, 1 << (PIO_CTRL_SM_ENABLE_LSB + 0));

```

Most instructions are executed from the instruction memory, but there are other sources, which can be freely mixed:

- Instructions written to a special configuration register (**SM_x INSTR**) are immediately executed, momentarily interrupting other execution. For example, a **JMP** instruction written to **SM_x INSTR** will cause the state machine to start executing from a different location.
- Instructions can be executed from a register, using the **MOV EXEC** instruction.
- Instructions can be executed from the output shifter, using the **OUT EXEC** instruction

The last of these is particularly versatile: instructions can be embedded in the stream of data passing through the FIFO. The I2C example uses this to embed e.g. **STOP** and **RESTART** line conditions alongside normal data. In the case of **MOV** and **OUT EXEC**, the **MOV/OUT** itself executes in one cycle, and the executee on the next.

3.2.3. Registers

Each state machine possesses a small number of internal registers. These hold input or output data, and temporary values such as loop counter variables.

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第42至47行

```

42 // 配置状态机0以sysclk/2.5的速度运行。状态机可
43 // 以每个时钟周期执行一条指令的最快速度运行，但我们可以统一
44 // 降低其速度以满足特定的精确频率需求，例如
45 // UART 波特率。该寄存器具有 16 位整数除数位和 8 位
46 // 小数除数位。
47 pio->sm[0].clkdiv = (uint32_t)(2.5f * (1 << 16));

```

上述代码片段属于完整示例代码，该示例通过 GPIO 0（或其他任意选择映射的引脚）输出12.5MHz方波。我们还可以使用pinsWAIT PIN指令暂停状态机执行一段时间，或使用JMP PIN指令依据引脚状态进行跳转，使控制流因引脚状态而异。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 51 至 59 行

```

51 // 引脚映射分为五组 (out、in、set、side-set、jmp pin)
52 // 这些在不同指令或不同情况下使用。
53 // 此处仅使用 SET 指令。配置状态机 0 的 SET 指令
54 // 仅作用于 GPIO 0；随后配置 GPIO0 由 PIO0 控制，
55 // 而非例如处理器。
56 pio->sm[0].pinctrl =
57     (1 << PIO_SM0_PINCTRL_SET_COUNT_LSB) |
58     (0 << PIO_SM0_PINCTRL_SET_BASE_LSB);
59 gpio_set_function(0, pio_get_funcsel(pio));

```

系统可通过 CTRL 寄存器随时启动或停止各状态机。多个状态机可同时启动，且 PIO 的确定性保证它们能保持精确同步。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.c> 第 63 至 67 行

```

63 // 启动状态机运行。PIO CTRL 寄存器在 PIO
64 // 实例中为全局，因此您可以启动/停止多个状态机
65 // 同时运行。我们利用寄存器的硬件原子设置别名
66 // 使某一位位高，而无需对寄存器进行读-改-写操作。
67 hw_set_bits(&pio->ctrl, 1 << (PIO_CTRL_SM_ENABLE_LSB + 0));

```

大多数指令从指令存储器执行，但也存在其他来源，且可自由组合：

- 写入特殊配置寄存器 (**SMx INSTR**) 的指令会被立即执行，暂时中断其他指令的执行。例如，写入**SMx INSTR**的**JMP**指令将导致状态机从不同位置开始执行。
- 指令也可通过**MOV EXEC**指令从寄存器执行。
- 指令也可通过**OUT EXEC**指令从输出移位器执行。

最后一个尤其多功能：指令可以嵌入通过FIFO传输的数据流中。

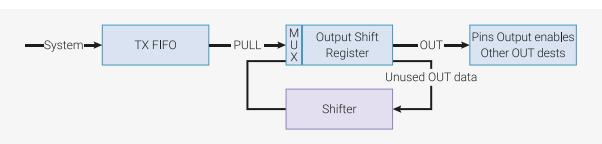
I2C示例利用此功能将例如 **STOP**和 **RESTART**线路状态嵌入到正常数据流中。对于 **MOV**和 **OUT EXEC**，**MOV/OUT**自身在一个周期内执行，执行对象则在下一个周期执行。

3.2.3. 寄存器

每个状态机均包含少量内部寄存器。这些寄存器用于保存输入或输出数据，以及临时值，如循环计数变量。

3.2.3.1. Output Shift Register (OSR)

Figure 40. Output Shift Register (OSR). Data is parcelled out 1...32 bits at a time, and unused data is recycled by a bidirectional shifter. Once empty, the OSR is reloaded from the TX FIFO.



The Output Shift Register (OSR) holds and shifts output data, between the TX FIFO and the pins (or other destinations, such as the scratch registers).

- **PULL** instructions: remove a 32-bit word from the TX FIFO and place into the OSR.
- **OUT** instructions shift data from the OSR to other destinations, 1...32 bits at a time.
- The OSR fills with zeroes as data is shifted out
- The state machine will automatically refill the OSR from the FIFO on an **OUT** instruction, once some total shift count threshold is reached, if autopull is enabled
- Shift direction can be left/right, configurable by the processor via configuration registers

For example, to stream data through the FIFO and output to the pins at a rate of one byte per two clocks:

```

1 .program pull_example1
2 loop:
3   out pins, 8
4 public entry_point:
5   pull
6   out pins, 8 [1]
7   out pins, 8 [1]
8   out pins, 8
9   jmp loop
  
```

Autopull (see [Section 3.5.4](#)) allows the hardware to automatically refill the OSR in the majority of cases, with the state machine stalling if it tries to **OUT** from an empty OSR. This has two benefits:

- No instructions spent on explicitly pulling from FIFO at the right time
- Higher throughput: can output up to 32 bits on every single clock cycle, if the FIFO stays topped up

After configuring autopull, the above program can be simplified to the following, which behaves identically:

```

1 .program pull_example2
2
3 loop:
4   out pins, 8
5 public entry_point:
6   jmp loop
  
```

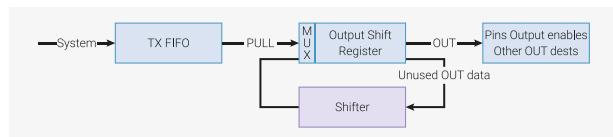
Program wrapping ([Section 3.5.2](#)) allows further simplification and, if desired, an output of 1 byte every system clock cycle.

```

1 .program pull_example3
2
3 public entry_point:
4 .wrap_target
5   out pins, 8 [1]
6 .wrap
  
```

3.2.3.1. 输出移位寄存器 (OSR)

图40。输出移位寄存器 (OSR)。数据以1至32位为单位发出，未使用的数据通过双向移位器予以回收。OSR一旦为空，即从TX FIFO重新加载数据。



输出移位寄存器 (OSR) 在 TX FIFO 与引脚（或其他目的地，如临时寄存器）之间保存并移位输出数据。

- **PULL** 指令：从 TX FIFO 中移除一个32位字并放入OSR。
- **OUT** 指令将数据从OSR移位至其他目的地，一次移位1至32位。
- 数据移出时，OSR会以零填充。
- 当启用autopull且达到某一总移位计数阈值时，状态机会在执行 **OUT**指令时自动从FIFO重新填充OSR。
- 移位方向可为左或右，由处理器通过配置寄存器进行配置。

例如，以下方法以每两个时钟周期输出一个字节的速率，通过FIFO传输数据至引脚：

```

1 .program pull_example1
2 loop:
3   out pins, 8
4 public entry_point:
5   pull
6   out pins, 8 [1]
7   out pins, 8 [1]
8   out pins, 8
9   jmp loop
  
```

Autopull（参见第3.5.4节）允许硬件在大多数情况下自动重新填充OSR；如果状态机尝试从空的OSR执行 **OUT**操作，则会暂停运行。这带来了两个好处：

- 无需指令在恰当时间显式地从FIFO中拉取数据
- 更高吞吐量：若FIFO保持充足，能在每个时钟周期输出最多32位

配置自动拉取后，上述程序可简化为如下内容，行为完全相同：

```

1 .program pull_example2
2
3 loop:
4   out pins, 8
5 public entry_point:
6   jmp loop
  
```

程序封装（第3.5.2节）允许进一步简化，且可实现每个系统时钟周期输出1字节。

```

1 .program pull_example3
2
3 public entry_point:
4 .wrap_target
5   out pins, 8 [1]
6 .wrap
  
```

3.2.3.2. Input Shift Register (ISR)

Figure 41. Input Shift Register (ISR). Data enters 1...32 bits at a time, and current contents is shifted left or right to make room. Once full, contents is written to the RX FIFO.



- **IN** instructions shift 1...32 bits at a time into the register.
- **PUSH** instructions write the ISR contents to the RX FIFO.
- The ISR is cleared to all-zeroes when pushed.
- The state machine will automatically push the ISR on an **IN** instruction, once some shift threshold is reached, if autopush is enabled.
- Shift direction is configurable by the processor via configuration registers

Some peripherals, like UARTs, must shift in from the left to get correct bit order, since the wire order is LSB-first; however, the processor may expect the resulting byte to be right-aligned. This is solved by the special **null** input source, which allows the programmer to shift some number of zeroes into the ISR, following the data.

3.2.3.3. Shift Counters

State machines remember how many bits, in total, have been shifted out of the OSR via **OUT** instructions, and into the **ISR** via **IN** instructions. This information is tracked at all times by a pair of hardware counters – the output shift counter and the input shift counter – each capable of holding values from 0 to 32 inclusive. With each shift operation, the relevant counter is incremented by the shift count, up to the maximum value of 32 (equal to the width of the shift register). The state machine can be configured to perform certain actions when a counter reaches a configurable threshold:

- The OSR can be automatically refilled once some number of bits have been shifted out. See [Section 3.5.4](#)
- The ISR can be automatically emptied once some number of bits have been shifted in. See [Section 3.5.4](#)
- **PUSH** or **PULL** instructions can be conditioned on the input or output shift counter, respectively

On PIO reset, or the assertion of **CTRL_SM_RESTART**, the input shift counter is cleared to 0 (nothing yet shifted in), and the output shift counter is initialised to 32 (nothing remaining to be shifted out; fully exhausted). Some other instructions affect the shift counters:

- A successful **PULL** clears the output shift counter to 0
- A successful **PUSH** clears the input shift counter to 0
- **MOV OSR, ...** (i.e. any **MOV** instruction that writes **OSR**) clears the output shift counter to 0
- **MOV ISR, ...** (i.e. any **MOV** instruction that writes **ISR**) clears the input shift counter to 0
- **OUT ISR, count** sets the input shift counter to **count**

3.2.3.4. Scratch Registers

Each state machine has two 32-bit internal scratch registers, called **X** and **Y**.

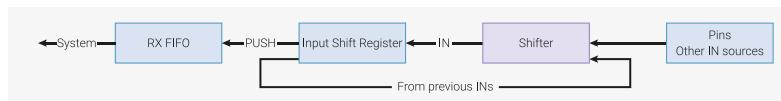
They are used as:

- Source/destination for **IN/OUT/SET/MOV**
- Source for branch conditions

For example, suppose we wanted to produce a long pulse for "1" data bits, and a short pulse for "0" data bits:

3.2.3.2. 输入移位寄存器 (ISR)

图41。输入移位寄存器 (ISR)。
数据一次输入1至32位，当前内容向左或向右移位以留出空间。
填满后，内容会写入接收FIFO。



- **IN** 指令一次将1至32位数据移入寄存器。
- **PUSH** 指令将ISR内容写入接收FIFO。
- ISR 在被推入时会被清零。
- 当启用自动推送功能且达到一定的移位阈值时，状态机会在执行 **IN** 指令时自动将ISR推送。
- 移位方向由处理器通过配置寄存器配置。

某些外设（如UART）必须从左侧移位以获得正确的位序列，因为线路顺序为最低有效位优先；然而，处理器可能期望得到的字节是右对齐的。此问题通过特殊的 **null** 输入源解决，该输入源允许程序员在数据之后向ISR中移入一定数量的零。

3.2.3.3. 移位计数器

状态机会记录通过 **OUT** 指令从OSR移出的位总数，以及通过 **IN** 指令移入ISR的位总数。此信息由一对硬件计数器持续跟踪——输出移位计数器和输入移位计数器——每个计数器均可保存从0至32（含）的数值。每次移位操作时，相应计数器按移位数量递增，最高可达32（等同于移位寄存器的宽度）。状态机可配置为在计数器达到设定阈值时执行特定操作：

- 当移出一定数量的位后，OSR可自动重新填充。详见第3.5.4节。
- 当移入一定数量的位后，ISR可自动清空。详见第3.5.4节。
- **PUSH** 或 **PULL** 指令分别可基于输入或输出移位计数器的状态进行条件执行。

在PIO复位或断言 **CTRL_SM_RESTART** 信号时，输入移位计数器清零（尚未移入任何位），输出移位计数器初始化为32（无剩余位需移出，已完全耗尽）。其他若干指令会影响移位计数器：

- 成功的 **PULL** 操作将输出移位计数器清零。
- 成功的 **PUSH** 操作将输入移位计数器清零。
- **MOV OSR, ...** （即任何写入 **OSR** 的 **MOV** 指令）将输出移位计数器清零为0
- **MOV ISR, ...** （即任何写入 **ISR** 的 **MOV** 指令）将输入移位计数器清零为0
- **OUT ISR, count** 将输入移位计数器设置为 **count**

3.2.3.4. 暂存寄存器

每个状态机均含两个32位内部暂存寄存器，称为 **X** 和 **Y**。

其用途包括：

- **IN/OUT/SET/MOV** 的源或目的地
- 分支条件的源

例如，假设我们希望为“1”数据位产生长脉冲，为“0”数据位产生短脉冲：

```

1 .program ws2812_led
2
3 public entry_point:
4     pull
5     set x, 23      ; Loop over 24 bits
6 bitloop:
7     set pins, 1      ; Drive pin high
8     out y, 1 [5]    ; Shift 1 bit out, and write it to y
9     jmp !y skip    ; Skip the extra delay if the bit was 0
10    nop [5]
11 skip:
12    set pins, 0 [5]
13    jmp x-- bitloop ; Jump if x nonzero, and decrement x
14    jmp entry_point

```

Here **X** is used as a loop counter, and **Y** is used as a temporary variable for branching on single bits from the OSR. This program can be used to drive a WS2812 LED interface, although more compact implementations are possible (as few as 3 instructions).

MOV allows the use of the scratch registers to save/restore the shift registers if, for example, you would like to repeatedly shift out the same sequence.

NOTE

A much more compact WS2812 example (4 instructions total) is shown in [Section 3.6.2](#)

3.2.3.5. FIFOs

Each state machine has a pair of 4-word deep FIFOs, one for data transfer from system to state machine (TX), and the other for state machine to system (RX). The TX FIFO is written to by system busmasters, such as a processor or DMA controller, and the RX FIFO is written to by the state machine. FIFOs decouple the timing of the PIO state machines and the system bus, allowing state machines to go for longer periods without processor intervention.

FIFOs also generate data request (DREQ) signals, which allow a system DMA controller to pace its reads/writes based on the presence of data in an RX FIFO, or space for new data in a TX FIFO. This allows a processor to set up a long transaction, potentially involving many kilobytes of data, which will proceed with no further processor intervention.

Often, a state machine is only transferring data in one direction. In this case the **SHIFTCTRL_FJOIN** option can merge the two FIFOs into a single 8-entry FIFO going in one direction only. This is useful for high-bandwidth interfaces such as DPI.

3.2.4. Stalling

State machines may momentarily pause execution for a number of reasons:

- A **WAIT** instruction's condition is not yet met
- A blocking **PULL** when the TX FIFO is empty, or a blocking **PUSH** when the RX FIFO is full
- An **IRQ WAIT** instruction which has set an IRQ flag, and is waiting for it to clear
- An **OUT** instruction when autopull is enabled, and OSR has already reached its shift threshold
- An **IN** instruction when autopush is enabled, ISR reaches its shift threshold, and the RX FIFO is full

In this case, the program counter does not advance, and the state machine will continue executing this instruction on the next cycle. If the instruction specifies some number of delay cycles before the next instruction starts, these do not begin until **after** the stall clears.

```

1 .program ws2812_led
2
3 public entry_point:
4     pull
5     set x, 23      ; 对24位进行循环
6 bitloop:
7     set pins, 1      ; 将引脚设为高电平
8     out y, 1 [5]    ; 将位向右移1位，并写入到y
9     jmp !y skip     ; 如果该位为0，则跳过额外延迟
10    nop [5]
11 skip:
12    设置引脚, 0 [5]
13    jmp x-- bitloop ; 如果x非零则跳转，并将x递减
14    jmp entry_point

```

此处 **X**用作循环计数器，**Y**用作分支时用于读取OSR中单个位的临时变量。该程序可用于驱动WS2812 LED接口，尽管存在更紧凑的实现方案（最少3条指令）。

MOV指令允许使用临时寄存器保存或恢复移位寄存器，以便例如重复移出相同序列。

i 注意

更紧凑的WS2812示例（共4条指令）见第3.6.2节

3.2.3.5. FIFO队列

每个状态机配备一对深度为4字的FIFO队列，分别用于系统向状态机的数据传输（TX）和状态机向系统的数据传输（RX）。TX FIFO由系统总线主控设备（如处理器或DMA控制器）写入，RX FIFO由状态机写入。FIFO实现了PIO状态机与系统总线的时序解耦，使状态机能够在更长时间内无需处理器干预而持续运行。

FIFO还会产生数据请求（DREQ）信号，允许系统DMA控制器根据RX FIFO中数据的存在或TX FIFO中可用空间的情况调整其读写节奏。这样，处理器可以设置一个可能涉及数千字节数据的长事务，事务执行过程中无需进一步的处理器干预。

通常状态机仅在单向进行数据传输。在此情况下，**SHIFTCTRL_FJOIN**选项可以将两个FIFO合并为单向的8级FIFO。此功能对于DPI等高带宽接口尤为重要。

3.2.4. 阻塞

状态机可能因多种原因而暂时暂停执行：

- 等待 **WAIT**指令的条件尚未满足
- 当TX FIFO为空时的阻塞 **PULL**，或当RX FIFO已满时的阻塞 **PUSH**
- 设置了IRQ标志且正在等待其清除的**IRQ WAIT**指令
- 启用自动拉取（**autopull**）且OSR已达到移位阈值时的 **OUT** 指令
- 启用自动推送（**autopush**），ISR达到移位阈值且RX FIFO已满时的 **IN** 指令

在此情况下，程序计数器不会前进，状态机将在下一周期继续执行该指令。若指令指定下一条指令开始前的延迟周期数，则该延迟将于阻塞解除后开始。

NOTE

Side-set ([Section 3.5.1](#)) is not affected by stalls, and always takes place on the first cycle of the attached instruction.

3.2.5. Pin Mapping

PIO controls the output level and direction of up to 32 GPIOs, and can observe their input levels. On every system clock cycle, each state machine may do none, one, or both of the following:

- Change the level or direction of some GPIOs via an `OUT` or `SET` instruction, or read some GPIOs via an `IN` instruction
- Change the level or direction of some GPIOs via a side-set operation

Each of these operations is on one of four contiguous ranges of GPIOs, with the base and count of each range configured via each state machine's `PINCTRL` register. There is a range for each of `OUT`, `SET`, `IN` and side-set operations. Each range can cover any of the GPIOs accessible to a given PIO block (on RP2040 this is the 30 user GPIOs), and the ranges can overlap.

For each individual GPIO output (level and direction separately), PIO considers all 8 writes that may have occurred on that cycle, and applies the write from the highest-numbered state machine. If the same state machine performs a `SET` /`OUT` and a side-set on the same GPIO simultaneously, the side-set is used. If no state machine writes to this GPIO output, its value does not change from the previous cycle.

Generally each state machine's outputs are mapped to a distinct group of GPIOs, implementing some peripheral interface.

3.2.6. IRQ Flags

IRQ flags are state bits which can be set or cleared by state machines or the system. There are 8 in total: all 8 are visible to all state machines, and the lower 4 can also be masked into one of PIO's interrupt request lines, via the `IRQ0_INTE` and `IRQ1_INTE` control registers.

They have two main uses:

- Asserting system level interrupts from a state machine program, and optionally waiting for the interrupt to be acknowledged
- Synchronising execution between two state machines

State machines interact with the flags via the `IRQ` and `WAIT` instructions.

3.2.7. Interactions Between State Machines

The instruction memory is implemented as a 1-write 4-read register file, so all four state machines can read an instruction on the same cycle, without stalling.

There are three ways to apply the multiple state machines:

- Pointing multiple state machines at the same program
- Pointing multiple state machines at different programs
- Using multiple state machines to run different parts of the same interface, e.g. TX and RX side of a UART, or clock/hsync and pixel data on a DPI display

State machines can not communicate data, but they can synchronise with one another by using the IRQ flags. There are 8 flags total (the lower four of which can be masked for use as system IRQs), and each state machine can set or clear any flag using the `IRQ` instruction, and can wait for a flag to go high or low using the `WAIT IRQ` instruction. This allows cycle-accurate synchronisation between state machines.

ⓘ 注意

侧置设置（第3.5.1节）不受阻塞影响，始终在附属指令的第一个周期执行。

3.2.5. 引脚映射

PIO可控制最多32个GPIO的输出电平和方向，并可检测其输入电平。在每个系统时钟周期内，每个状态机可执行以下操作中的零项、一项或两项：

- 通过 `OUT` 或 `SET` 指令更改部分GPIO的电平或方向，或通过 `IN` 指令读取部分GPIO
- 通过副设置操作更改部分GPIO的电平或方向

上述每项操作均作用于四个连续GPIO范围之一，该范围的起始地址和数量由各状态机的 `PINCTRL` 寄存器配置。每种操作——`OUT`、`SET`、`IN` 及副设置——各对应一个范围。

每个范围可覆盖特定PIO模块可访问的任意GPIO（RP2040上为30个用户GPIO），且范围可相互重叠。

对于每个独立GPIO的输出（电平和方向分别处理），PIO会综合当周期内的最多8次写操作，应用编号最高的状态机的写入。若同一状态机在同一GPIO同时执行 `SET /OUT` 和副设置，则采用副设置操作。如果没有状态机写入此GPIO输出，其值将保持上一个周期的状态不变。

通常，每个状态机的输出映射到一组独立的GPIO，以实现某种外设接口。

3.2.6. IRQ 标志

IRQ标志是状态位，可以由状态机或系统设置或清除。共有8个：全部8个对所有状态机可见，且低4位可通过 `IRQ0_INTE` 和 `RQ1_INTE` 控制寄存器掩码至PIO的中断请求线上。

它们具有两个主要用途：

- 由状态机程序断言系统级中断，并可选择性地等待中断被确认
- 同步两个状态机之间的执行

状态机通过 `IRQ` 和 `WAIT` 指令与这些标志进行交互。

3.2.7. 状态机之间的交互

指令存储器实现为1写4读寄存器文件，因此所有四个状态机可在同一周期读取指令，无需停顿。

多状态机的应用方式有三种：

- 将多个状态机指向同一程序
- 将多个状态机指向不同程序
- 使用多个状态机运行同一接口的不同部分，例如UART的发送（TX）与接收（RX）端，或DPI显示上的时钟/行同步信号与像素数据

状态机之间无法传递数据，但可通过IRQ标志进行同步。共有8个标志（低四位可屏蔽，用作系统IRQ），每个状态机可通过 `IRQ` 指令设置或清除任一标志，并可利用 `WAIT IRQ` 指令等待标志变高或变低。此机制实现了状态机之间的周期精确同步。

3.3. PIO Assembler (pioasm)

The PIO Assembler parses a PIO source file and outputs the assembled version ready for inclusion in an RP2040 application. This includes C and C++ applications built against the SDK, and Python programs running on the RP2040 MicroPython port.

This section briefly introduces the directives and instructions that can be used in `pioasm` input. A deeper discussion of how to use `pioasm`, how it is integrated into the SDK build system, extended features such as code pass through, and the various output formats it can produce, is given in the [Raspberry Pi Pico-series C/C++ SDK](#) book.

3.3.1. Directives

The following directives control the assembly of PIO programs:

Table 363. pioasm directives

<code>.define (PUBLIC) <symbol> <value></code>	Define an integer symbol named <code><symbol></code> with the value <code><value></code> (see Section 3.3.2). If this <code>.define</code> appears before the first program in the input file, then the define is global to all programs, otherwise it is local to the program in which it occurs. If <code>PUBLIC</code> is specified the symbol will be emitted into the assembled output for use by user code. For the SDK this takes the form of: <code>#define <program_name>_<symbol> value</code> for program symbols or <code>#define <symbol> value</code> for global symbols
<code>.program <name></code>	Start a new program with the name <code><name></code> . Note that that name is used in code so should be alphanumeric/underscore not starting with a digit. The program lasts until another <code>.program</code> directive or the end of the source file. PIO instructions are only allowed within a program
<code>.origin <offset></code>	Optional directive to specify the PIO instruction memory offset at which the program <i>must</i> load. Most commonly this is used for programs that must load at offset 0, because they use data based JMPs with the (absolute) jmp target being stored in only a few bits. This directive is invalid outside of a program
<code>.side_set <count> (opt) (pindirs)</code>	If this directive is present, <code><count></code> indicates the number of side-set bits to be used. Additionally <code>opt</code> may be specified to indicate that a <code>side <value></code> is optional for instructions (note this requires stealing an extra bit – in addition to the <code><count></code> bits – from those available for the instruction delay). Finally, <code>pindirs</code> may be specified to indicate that the side set values should be applied to the PINDIRs and not the PINs. This directive is only valid within a program before the first instruction
<code>.wrap_target</code>	Place prior to an instruction, this directive specifies the instruction where execution continues due to program wrapping. This directive is invalid outside of a program, may only be used once within a program, and if not specified defaults to the start of the program
<code>.wrap</code>	Placed after an instruction, this directive specifies the instruction after which, in normal control flow (i.e. <code>jmp</code> with false condition, or no <code>jmp</code>), the program wraps (to <code>.wrap_target</code> instruction). This directive is invalid outside of a program, may only be used once within a program, and if not specified defaults to after the last program instruction.
<code>.lang_opt <lang> <name> <option></code>	Specifies an option for the program related to a particular language generator. (See Language generators). This directive is invalid outside of a program
<code>.word <value></code>	Stores a raw 16-bit value as an instruction in the program. This directive is invalid outside of a program.

3.3. PIO 汇编器 (pioasm)

PIO汇编器解析PIO源文件，输出已汇编的版本，供包含于RP2040应用程序中使用。包括基于SDK构建的C和C++应用程序，以及运行于RP2040 MicroPython移植版的Python程序。

本节简要介绍可用于 `pioasm` 输入的指令及指示。关于如何使用 `pioasm`、其在SDK构建系统中的集成方式、扩展功能（如代码穿透）以及其支持的多种输出格式的深入讨论，请参见[Raspberry Pi Pico系列C/C++ SDK手册](#)。

3.3.1. 指令

以下指令用于控制PIO程序的汇编：

表363. `pioasm`
指令

<code>.define (PUBLIC)<symbol><value></code>	定义一个名为 <code><symbol></code> 的整型符号，其值为 <code><value></code> （ 参见第3.3.2节 ）。若此 <code>.define</code> 位于输入文件中第一个程序之前，则定义对所有程序全局有效，否则仅对定义所在的程序局部有效。若指定了 <code>PUBLIC</code> ，该符号将被包含在汇编输出中，以供用户代码使用。对于SDK，其形式如下：
<code>#define <program_name>_<symbol> value</code>	用于程序符号，或 <code>#define <symbol> value</code> 用于全局符号
<code>.program <name></code>	启动一个名为 <code><name></code> 的新程序。请注意，该名称用于代码中，应为字母数字或下划线组成且不能以数字开头。程序将持续，直到遇到另一个 <code>.program</code> 指令或源文件结束。PIO 指令仅允许在程序内使用。
<code>.origin <offset></code>	可选指令，用于指定程序必须加载到的PIO指令存储偏移地址。此指令最常用于必须从偏移量0加载的程序，因为它们使用基于数据的JMP，其跳转目标（绝对地址）仅存储在少数几位中。此指令在程序外部无效。
<code>.side_set <count> (opt) (pindirs)</code>	如果存在此指令， <code><count></code> 表示所使用的边置（side-set）位数。此外， <code>opt</code> 可用于指示指令中的边置 <code><value></code> 为可选（注意，这需要额外占用一位 —— 除了 <code><count></code> 位之外 —— 从指令延迟的可用位中借用）。最后， <code>pindirs</code> 可用于指定侧集值应应用于PINDIRs而非PINs。此指令仅在程序内且首条指令之前有效。
<code>.wrap_target</code>	置于指令之前，该指令指定程序因环绕而继续执行的指令位置。此指令在程序外无效，程序内仅可使用一次；若未指定，默认指向程序起始位置。
<code>.wrap</code>	置于指令之后，该指令指定在正常控制流（即条件为假时的 <code>jmp</code> 或无 <code>jmp</code> ）下，程序环绕至 <code>.wrap_target</code> 指令之前的位置。此指令在程序外无效，程序内仅可使用一次；若未指定，默认指向最后一条程序指令之后。
<code>.lang_opt <lang> <name> <option></code>	指定与特定语言生成器相关的程序选项。 (参见语言生成器) 此指令在程序外部无效。
<code>.word <value></code>	将一个原始16位数值作为程序中的指令存储。该指令在程序外无效。

3.3.2. Values

The following types of values can be used to define integer numbers or branch targets

Table 364. Values in pioasm, i.e. <value>

<code>integer</code>	An integer value e.g. 3 or -7
<code>hex</code>	A hexadecimal value e.g. <code>0xf</code>
<code>binary</code>	A binary value e.g. <code>0b1001</code>
<code>symbol</code>	A value defined by a <code>.define</code> (see pioasm_define)
<code><label></code>	The instruction offset of the label within the program. This makes most sense when used with a JMP instruction (see Section 3.4.2)
<code>(<expression>)</code>	An expression to be evaluated; see expressions . Note that the parentheses are necessary.

3.3.3. Expressions

Expressions may be freely used within pioasm values.

Table 365. Expressions in pioasm i.e. <expression>

<code><expression> + <expression></code>	The sum of two expressions
<code><expression> - <expression></code>	The difference of two expressions
<code><expression> * <expression></code>	The multiplication of two expressions
<code><expression> / <expression></code>	The integer division of two expressions
<code>- <expression></code>	The negation of another expression
<code>:: <expression></code>	The bit reverse of another expression
<code><value></code>	Any value (see Section 3.3.2)

3.3.4. Comments

Line comments are supported with `//` or `;`

C-style block comments are supported via `/*` and `*/`

3.3.5. Labels

Labels are of the form:

`<symbol>:`

or

`PUBLIC <symbol>:`

at the start of a line.

3.3.2. 值

以下类型的数值可用于定义整数或分支目标：

表364。pioasm 中的数值，即 `<value>`

整数	整数值，例如 3 或 -7
十六进制	十六进制数值，例如 <code>0xf</code>
二进制	二进制数值，例如 <code>0b1001</code>
符号	由 <code>.define</code> 定义的值（参见 <code>pioasm_define</code> ）
<code><label></code>	标签在程序中的指令偏移量。通常与 JMP 指令一同使用时最为合理（参见第 3.4.2 节）
<code>(<expression>)</code>	待求值的表达式；详见表达式。请注意，括号为必需。

3.3.3. 表达式

表达式可自由用于 pioasm 的值中。

表 365。
pioasm 中的表达式，即 `<expression>`

<code><expression> + <expression></code>	两个表达式之和
<code><expression> - <expression></code>	两个表达式之差
<code><expression> * <expression></code>	两个表达式的乘积
<code><expression> / <expression></code>	两个表达式的整数除法
<code>- <expression></code>	另一个表达式的取反
<code>:: <expression></code>	另一个表达式的位反转
<code><value></code>	任意数值（参见第3.3.2节）

3.3.4. 注释

支持使用 `//` 或 `;进行行注释`

支持使用 `/*` 和 `*/` 进行 C 风格块注释

3.3.5. 标签

标签格式如下：

`<symbol>:`

或

`PUBLIC <symbol>:`

位于行首。

TIP

A label is really just an automatic `.define` with a value set to the current program instruction offset. A *PUBLIC* label is exposed to the user code in the same way as a *PUBLIC* `.define`.

3.3.6. Instructions

All pioasm instructions follow a common pattern:

`<instruction> (side <side_set_value>) ({<delay_value>})`

where:

- `<instruction>` Is an assembly instruction detailed in the following sections. (See [Section 3.4](#))
- `<side_set_value>` Is a value (see [Section 3.3.2](#)) to apply to the side_set pins at the start of the instruction. Note that the rules for a side-set value via `side <side_set_value>` are dependent on the `.side_set` (see `pioasm_side_set`) directive for the program. If no `.side_set` is specified then the `side <side_set_value>` is invalid, if an optional number of sideset pins is specified then `side <side_set_value>` may be present, and if a non-optional number of sideset pins is specified, then `side <side_set_value>` is required. The `<side_set_value>` must fit within the number of side-set bits specified in the `.side_set` directive.
- `<delay_value>` Specifies the number of cycles to delay after the instruction completes. The `delay_value` is specified as a value (see [Section 3.3.2](#)), and in general is between 0 and 31 inclusive (a 5-bit value), however the number of bits is reduced when sideset is enabled via the `.side_set` (see `pioasm_side_set`) directive. If the `<delay_value>` is not present, then the instruction has no delay

NOTE

pioasm instruction names, keywords and directives are case insensitive; lower case is used in the Assembly Syntax sections below as this is the style used in the SDK.

NOTE

Commas appear in some Assembly Syntax sections below, but are entirely optional, e.g. `out pins, 3` may be written `out pins 3`, and `jmp x-- label` may be written as `jmp x--, label`. The Assembly Syntax sections below uses the first style in each case as this is the style used in the SDK.

3.3.7. Pseudoinstructions

Currently pioasm provides one pseudoinstruction, as a convenience:

- `nop` Assembles to `mov y, y`. "No operation", has no particular side effect, but a useful vehicle for a side-set operation or an extra delay.

3.4. Instruction Set

3.4.1. Summary

PIO instructions are 16 bits long, and have the following encoding:

💡 提示

标签实际上是一个自动生成的 `.define`，其值设为当前程序指令偏移量。PUBLIC标签以与PUBLIC.define相同的方式向用户代码公开。

3.3.6. 指令

所有pioasm指令遵循以下通用格式：

`<instruction> (side <side_set_value>) ([<delay_value>])`

其中：

`<instruction>` 为后续章节详细说明的汇编指令。（参见第3.4节）

`<side_set_value>` 为指令开始时应用于side_set引脚的数值（参见第3.3.2节）。请注意，通过`side <side_set_value>`设置的side-set值规则，取决于程序的`.side_set`（参见pioasm_side_set）指令。如果未指定`side_set`，则侧面`<side_set_value>`无效；如果指定为可选数量的 sideset 引脚，则侧面`<side_set_value>`可能存在；如果指定为非可选数量的 sideset 引脚，则侧面`<side_set_value>`是必需的。`<side_set_value>`必须在`.side_set`中指定的 sideset 位数范围内。

指令。

`<delay_value>` 指定指令执行完成后的延迟周期数。delay_value 以数值形式指定（详见第 3.3.2 节），通常取值范围为 0 至 31（含）（5 位值），但启用`.side_set`（详见 pioasm_side_set）指令时，位数会相应减少。如果未提供`<delay_value>`，则指令无延迟。

💡 注意

pioasm 的指令名称、关键字及指令对大小写不敏感；下文的汇编语法部分使用小写字母，因为这符合SDK的风格。

💡 注意

部分汇编语法章节中出现逗号，但完全可选，例如：out pins, 3 可写作out pins 3，jmp x-- /label/ 可写作jmp x--, /label/。下文汇编语法部分均采用第一种风格，因为这符合SDK的用法。

3.3.7. 伪指令

pioasm目前提供一条便捷的伪指令：

`nop`汇编为`mov y, y`。“无操作”指令无特殊副作用，但常用作side-set操作或额外延迟的工具。

3.4. 指令集

3.4.1. 概要

PIO指令长度为16位，其编码如下：

Table 366. PIO instruction encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
JMP	0	0	0	Delay/side-set					Condition			Address								
WAIT	0	0	1	Delay/side-set					Pol	Source		Index								
IN	0	1	0	Delay/side-set					Source			Bit count								
OUT	0	1	1	Delay/side-set					Destination			Bit count								
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0				
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0				
MOV	1	0	1	Delay/side-set					Destination			Op	Source							
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index								
SET	1	1	1	Delay/side-set					Destination			Data								

All PIO instructions execute in one clock cycle.

The function of the 5-bit `Delay/side-set` field depends on the state machine's `SIDESET_COUNT` configuration:

- Up to 5 LSBs (5 minus `SIDESET_COUNT`) encode a number of idle cycles inserted between this instruction and the next.
- Up to 5 MSBs, set by `SIDESET_COUNT`, encode a side-set (Section 3.5.1), which can assert a constant onto some GPIOs, concurrently with main instruction execution.

3.4.2. JMP

3.4.2.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				

3.4.2.2. Operation

Set program counter to `Address` if `Condition` is true, otherwise no operation.

Delay cycles on a `JMP` always take effect, whether `Condition` is true or false, and they take place *after* `Condition` is evaluated and the program counter is updated.

- Condition:
 - 000: (*no condition*): Always
 - 001: `!X`: scratch X zero
 - 010: `X--`: scratch X non-zero, prior to decrement
 - 011: `!Y`: scratch Y zero
 - 100: `Y--`: scratch Y non-zero, prior to decrement
 - 101: `X!=Y`: scratch X not equal scratch Y
 - 110: `PIN`: branch on input pin
 - 111: `!OSRE`: output shift register not empty
- Address: Instruction address to jump to. In the instruction encoding this is an absolute address within the PIO instruction memory.

表366. PIO
指令编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
JMP	0	0	0	延迟/副控					条件			地址					
等待	0	0	1	延迟/副控					极性	来源		索引					
输入	0	1	0	延迟/副控					来源			位数					
输出	0	1	1	延迟/副控					目标			位数					
推送	1	0	0	延迟/副控					0	IfF	块	0	0	0	0	0	
拉取	1	0	0	延迟/副控					1	IfE	块	0	0	0	0	0	
MOV	1	0	1	延迟/副控					目标			操作		来源			
IRQ	1	1	0	延迟/副控					0	清除	等待	索引					
设置	1	1	1	延迟/副控					目标			数据					

所有 PIO 指令均在一个时钟周期内完成执行。

5 位延迟/侧置字段的功能取决于状态机的SIDESET_COUNT 配置：

- 最多 5 个最低有效位（5 减去SIDESET_COUNT）用于编码在本指令与下一指令之间插入的空闲周期数。
- 最多 5 个最高有效位，由SIDESET_COUNT设置，用于编码侧置（第 3.5.1 节），可在主指令执行的同时向部分 G PIO 置入常量。

3.4.2. JMP

3.4.2.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	延迟/副控					条件			地址				

3.4.2.2. 操作

如果条件为真，则将程序计数器设置为地址，否则无操作。

延迟周期在 JMP 指令中始终生效，无论条件为真还是假，且发生在条件被评估及程序计数器更新之后。

- 条件：
 - 000: (无条件)：始终成立
 - 001: !X: 寄存器X 为零
 - 010: X--: 寄存器X 非零，递减前
 - 011: !Y: 寄存器Y 为零
 - 100: Y--: 寄存器Y 非零，递减前
 - 101: X!=Y: 寄存器X 不等于寄存器Y
 - 110: PIN: 根据输入引脚分支
 - 111: !OSRE: 输出移位寄存器非空
- 地址：跳转的指令地址。在指令编码中，该地址为PIO指令存储器内的绝对地址。

`JMP PIN` branches on the GPIO selected by `EXECCTRL JMP PIN`, a configuration field which selects one out of the maximum of 32 GPIO inputs visible to a state machine, independently of the state machine's other input mapping. The branch is taken if the GPIO is high.

`!OSRE` compares the bits shifted out since the last `PULL` with the shift count threshold configured by `SHIFTCTRL_PULL_THRESH`. This is the same threshold used by autopull ([Section 3.5.4](#)).

`JMP X--` and `JMP Y--` always decrement scratch register X or Y, respectively. The decrement is not conditional on the current value of the scratch register. The branch is conditioned on the *initial* value of the register, i.e. before the decrement took place: if the register is initially nonzero, the branch is taken.

3.4.2.3. Assembler Syntax

`jmp (<cond>) <target>`

where:

- <cond> Is an optional condition listed above (e.g. `!x` for scratch X zero). If a condition code is not specified, the branch is always taken
- <target> Is a program label or value (see [Section 3.3.2](#)) representing instruction offset within the program (the first instruction being offset 0). Note that because the PIO JMP instruction uses absolute addresses in the PIO instruction memory, JMPs need to be adjusted based on the program load offset at runtime. This is handled for you when loading a program with the SDK, but care should be taken when encoding JMP instructions for use by `OUT EXEC`

3.4.3. WAIT

3.4.3.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<code>WAIT</code>	0	0	1	Delay/side-set				Pol	Source		Index					

3.4.3.2. Operation

Stall until some condition is met.

Like all stalling instructions ([Section 3.2.4](#)), delay cycles begin after the instruction completes. That is, if any delay cycles are present, they do not begin counting until *after* the wait condition is met.

- Polarity:
 - 1: wait for a 1.
 - 0: wait for a 0.
- Source: what to wait on. Values are:
 - 00: `GPIO`: System GPIO input selected by `Index`. This is an absolute GPIO index, and is not affected by the state machine's input IO mapping.
 - 01: `PIN`: Input pin selected by `Index`. This state machine's input IO mapping is applied first, and then `Index` selects which of the mapped bits to wait on. In other words, the pin is selected by adding `Index` to the `PINCTRL_IN_BASE` configuration, modulo 32.
 - 10: `IRQ`: PIO IRQ flag selected by `Index`

JMP PIN根据`EXECCTRL JMP_PIN`选择的GPIO进行分支，该配置字段从最多32个状态机可见的GPIO输入中选择一个，独立于状态机的其他输入映射。当GPIO为高电平时，分支被采纳。

!OSRE 比较自上次 **PULL**以来移出的位与由`SHIFTCTRL_PULL_THRESH`配置的移位计数阈值。
该阈值与自动拉取（第3.5.4节）所用阈值相同。

JMP X--和 **JMP Y--**分别始终递减暂存寄存器X或Y的值。递减操作不依赖于暂存寄存器当前的值。分支条件基于寄存器的初始值，即递减前的值：若寄存器初始非零，则分支成立。

3.4.2.3. 汇编器语法

`jmp (<cond>) <target>`

其中：

<cond> 为上述可选条件之一（例如 `!x` 表示暂存寄存器X为零）。若未指定条件码，则分支总是被采纳。

<target> 为程序标签或值（参见第3.3.2节），表示程序内指令偏移（第一条指令偏移为0）。请注意，因PIO JMP指令使用PIO指令存储器中的绝对地址，JMP指令需根据程序运行时的加载偏移进行调整。SDK加载程序时会自动处理此问题，但编码供`OUT EXEC`使用的JMP指令时应格外注意。

3.4.3. WAIT

3.4.3.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
等待	0	0	1	延迟/副控				极性	来源		索引					

3.4.3.2. 操作

暂停执行，直到满足特定条件。

如所有暂停指令（第3.2.4节所述）一样，延迟周期自指令完成后开始计数。换言之，若存在延迟周期，其计数不会在等待条件满足之前开始。

- 极性：
 - 1: 等待值为1。
 - 0: 等待值为0。
- 来源：等待的对象。取值如下：
 - 00: **GPIO**: 由 `Index`指定的系统GPIO输入。该GPIO索引为绝对值，不受状态机输入IO映射影响。
 - 01: **PIN**: 由 `Index`选择的输入引脚。该状态机的输入IO映射先行应用，随后由 `Index`选择映射位中等待的具体位。换言之，选定的引脚是通过将`Index`加至`PINCTRL_IN_BASE`配置，取模32计算得出。
 - 10: **IRQ**: 由 `Index`选择的PIO IRQ标志

- 11: Reserved
- Index: which pin or bit to check.

`WAIT x IRQ` behaves slightly differently from other `WAIT` sources:

- If `Polarity` is 1, the selected IRQ flag is cleared by the state machine upon the wait condition being met.
- The flag index is decoded in the same way as the `IRQ` index field: if the MSB is set, the state machine ID (0...3) is added to the IRQ index, by way of modulo-4 addition on the two LSBs. For example, state machine 2 with a flag value of '0x11' will wait on flag 3, and a flag value of '0x13' will wait on flag 1. This allows multiple state machines running the same program to synchronise with each other.

⚠ CAUTION

`WAIT 1 IRQ x` should not be used with IRQ flags presented to the interrupt controller, to avoid a race condition with a system interrupt handler

3.4.3.3. Assembler Syntax

`wait <polarity> gpio <gpio_num>`

`wait <polarity> pin <pin_num>`

`wait <polarity> irq <irq_num> (rel)`

where:

- | | |
|--------------------------------------|---|
| <code><polarity></code> | Is a value (see Section 3.3.2) specifying the polarity (either 0 or 1) |
| <code><pin_num></code> | Is a value (see Section 3.3.2) specifying the input pin number (as mapped by the SM input pin mapping) |
| <code><gpio_num></code> | Is a value (see Section 3.3.2) specifying the actual GPIO pin number |
| <code><irq_num> (rel)</code> | Is a value (see Section 3.3.2) specifying The irq number to wait on (0-7). If <code>rel</code> is present, then the actual irq number used is calculating by replacing the low two bits of the irq number (irq_num_{10}) with the low two bits of the sum ($irq_num_{10} + sm_num_{10}$) where sm_num_{10} is the state machine number |

3.4.4. IN

3.4.4.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<code>IN</code>	0	1	0	Delay/side-set				Source			Bit count					

3.4.4.2. Operation

Shift `Bit count` bits from `Source` into the Input Shift Register (ISR). Shift direction is configured for each state machine by `SHIFTCTRL_IN_SHIFTDIR`. Additionally, increase the input shift count by `Bit count`, saturating at 32.

- Source:
 - 000: `PINS`
 - 001: `X` (scratch register X)

- 11: 保留
 - Index: 要检查的引脚或位。
- WAIT x IRQ** 的行为与其他 **WAIT** 源存在细微差异：
- 若 **Polarity** 为 1，则状态机在等待条件满足时清除所选的 IRQ 标志。
 - 标志索引的解码方式与 **IRQ** 索引字段一致：若最高有效位（MSB）被设置，则状态机 ID（0...3）通过对最低两位执行模4加法的方式加到 IRQ 索引上。例如，状态机 2 对于标志值“0x11”将等待标志 3，标志值“0x13”将等待标志 1。此功能允许多个运行相同程序的状态机实现相互同步。

⚠ 注意

WAIT 1 IRQ x 不应与提交给中断控制器的 IRQ 标志一起使用，以避免与系统中断处理程序产生竞态条件。

3.4.3.3. 汇编语法

```
wait <polarity> gpio <gpio_num>
wait <polarity> pin <pin_num>
wait <polarity> irq <irq_num> ( rel )
```

其中：

<polarity>	为一个值（详见第3.3.2节），指定极性（0或1）
<pin_num>	为一个值（详见第3.3.2节），指定输入引脚号（根据SM输入引脚映射）
<gpio_num>	为一个值（详见第3.3.2节），指定实际GPIO引脚号
<irq_num> (rel)	为一个值（详见第3.3.2节），指定等待的IRQ编号（0-7）。如果 rel 存在，则实际使用的 irq 号通过将 irq 号 (irq_num_{10}) 的低两位替换为和 ($irq_num_{10} + sm_num_{10}$) 低两位计算得出，其中 sm_num_{10} 为状态机编号。 数字

3.4.4. IN

3.4.4.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
输入	0	1	0		延迟/副控				来源						位数	

3.4.4.2. 操作

将位数位从源移入输入移位寄存器（ISR）。每个状态机的移位方向由 **SIFTCTRL_IN_SHIFTDIR** 配置。此外，将输入移位计数增加位数，最大值饱和为 32。

- 来源：
 - 000: 引脚
 - 001: X (临时寄存器 X)

- 010: **Y** (scratch register Y)
 - 011: **NULL** (all zeroes)
 - 100: Reserved
 - 101: Reserved
 - 110: **ISR**
 - 111: **OSR**
- Bit count: How many bits to shift into the ISR. 1...32 bits, 32 is encoded as **0000**.

If automatic push is enabled, **IN** will also push the ISR contents to the RX FIFO if the push threshold is reached (**SHIFTCTRL_PUSH_THRESH**). **IN** still executes in one cycle, whether an automatic push takes place or not. The state machine will stall if the RX FIFO is full when an automatic push occurs. An automatic push clears the ISR contents to all-zeroes, and clears the input shift count. See [Section 3.5.4](#).

IN always uses the least significant **Bit count** bits of the source data. For example, if **PINCTRL_IN_BASE** is set to 5, the instruction **IN PINS, 3** will take the values of pins 5, 6 and 7, and shift these into the ISR. First the ISR is shifted to the left or right to make room for the new input data, then the input data is copied into the gap this leaves. The bit order of the input data is not dependent on the shift direction.

NULL can be used for shifting the ISR's contents. For example, UARTs receive the LSB first, so must shift to the right. After 8 **IN PINS, 1** instructions, the input serial data will occupy bits 31...24 of the ISR. An **IN NULL, 24** instruction will shift in 24 zero bits, aligning the input data at ISR bits 7...0. Alternatively, the processor or DMA could perform a byte read from FIFO address + 3, which would take bits 31...24 of the FIFO contents.

3.4.4.3. Assembler Syntax

in <source>, <bit_count>

where:

<source> Is one of the sources specified above.

<bit_count> Is a value (see [Section 3.3.2](#)) specifying the number of bits to shift (valid range 1-32)

3.4.5. OUT

3.4.5.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT	0	1	1	Delay/side-set				Destination			Bit count					

3.4.5.2. Operation

Shift **Bit count** bits out of the Output Shift Register (OSR), and write those bits to **Destination**. Additionally, increase the output shift count by **Bit count**, saturating at 32.

- Destination:
 - 000: **PINS**
 - 001: **X** (scratch register X)
 - 010: **Y** (scratch register Y)

- 010: **Y** (临时寄存器 Y)
 - 011: 无效 (全零)
 - 100: 保留
 - 101: 保留
 - 110: **ISR**
 - 111: **OSR**
- 位数: 要移入 ISR 的比特数。取值范围为 1 至 32, 32 以 **00000** 编码。

如果启用了自动推送，**IN**将在达到推送阈值 (**SHIFTCTRL_PUSH_THRESH**) 时，将ISR内容推送至RX FIFO。无论是否发生自动推送，IN均在一个周期内执行。自动推送发生时若RX FIFO已满，状态机将进入阻塞状态。自动推送会将ISR内容清零，并重置输入移位计数。详见第3.5.4节。

IN始终使用源数据的最低有效位 (**Bit count** 位)。例如，若PINCTRL_IN_BASE设置为5，指令**IN PINS, 3**将采集第5、6、7号引脚的值，并将其移入ISR。首先ISR向左或向右移位以腾出存放新输入数据的空间，随后将输入数据复制至该空隙。输入数据的位序不依赖于移位方向。

NULL 可用于对ISR内容进行移位。例如，UART接收最低有效位 (LSB) 优先，因此必须向右移位。

经过8次输入引脚，1条指令后，输入的串行数据将占据ISR的31...24位。一次输入**NULL**，24条指令将移入24个零位，使输入数据对齐至ISR的7...0位。或者，处理器或DMA可以从FIFO地址 + 3执行字节读取，该操作将获取FIFO内容的31...24位。

3.4.4.3. 汇编语法

in <source>, <bit_count>

其中：

<*source*> 为上述指定的某一源。

<*bit_count*> 为一个值（参见第3.3.2节），指定移位的位数（有效范围为1至32位）

3.4.5. OUT

3.4.5.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
输出	0	1	1	延迟/副控				目标			位数					

3.4.5.2. 操作

Shift **Bit count** 位从输出移位寄存器 (OSR) 移出，并将这些位写入 **Destination**。此外，输出移位计数增加 **Bit count**，最大饱和至32。

- 目的地：

- 000: 引脚
- 001: **X** (临时寄存器 X)
- 010: **Y** (临时寄存器 Y)

- 011: **NULL** (discard data)
- 100: **PINDIRS**
- 101: **PC**
- 110: **ISR** (also sets ISR shift counter to **Bit count**)
- 111: **EXEC** (Execute OSR shift data as instruction)
- Bit count: how many bits to shift out of the OSR. 1...32 bits, 32 is encoded as **0000**.

A 32-bit value is written to **Destination**: the lower **Bit count** bits come from the OSR, and the remainder are zeroes. This value is the least significant **Bit count** bits of the OSR if **SHIFTCTRL_OUT_SHIFTDIR** is to the right, otherwise it is the most significant bits.

PINS and **PINDIRS** use the **OUT** pin mapping, as described in [Section 3.5.6](#).

If automatic pull is enabled, the OSR is automatically refilled from the TX FIFO if the pull threshold, **SHIFTCTRL_PULL_THRESH**, is reached. The output shift count is simultaneously cleared to 0. In this case, the **OUT** will stall if the TX FIFO is empty, but otherwise still executes in one cycle. The specifics are given in [Section 3.5.4](#).

OUT EXEC allows instructions to be included inline in the FIFO datastream. The **OUT** itself executes on one cycle, and the instruction from the OSR is executed on the next cycle. There are no restrictions on the types of instructions which can be executed by this mechanism. Delay cycles on the initial **OUT** are ignored, but the executee may insert delay cycles as normal.

OUT PC behaves as an unconditional jump to an address shifted out from the OSR.

3.4.5.3. Assembler Syntax

`out <destination>, <bit_count>`

where:

<destination> Is one of the destinations specified above.

<bit_count> Is a value (see [Section 3.3.2](#)) specifying the number of bits to shift (valid range 1-32)

3.4.6. PUSH

3.4.6.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH	1	0	0	Delay/side-set				0	IfF	Blk	0	0	0	0	0	0

3.4.6.2. Operation

Push the contents of the ISR into the RX FIFO, as a single 32-bit word. Clear ISR to all-zeroes.

- **IfFull**: If 1, do nothing unless the total input shift count has reached its threshold, **SHIFTCTRL_PUSH_THRESH** (the same as for autopush; see [Section 3.5.4](#)).
- **Block**: If 1, stall execution if RX FIFO is full.

PUSH IFFULL helps to make programs more compact, like autopush. It is useful in cases where the **IN** would stall at an inappropriate time if autopush were enabled, e.g. if the state machine is asserting some external control signal at this point.

- 011: **NULL** (丢弃数据)
- 100: **PINDIRS**
- 101: **PC**
- 110: **ISR** (同时将ISR移位计数器设为**Bit count**)
- 111: **EXEC** (将OSR移位数据作为指令执行)
- **Bit count**: 要从OSR移出的位数，范围为1至32位，32位编码为**00000**。

一个32位值被写入**Destination**: 低位的**Bit count**位取自OSR，其余位填充为零。该值为OSR的最低有效 **Bit count**位（当**SHIFTCTRL_OUT_SHIFTDIR**为向右时），否则为最高有效位。

PINS 和 **PINDIRS**采用第3.5.6节所述的 **OUT**引脚映射。

如果启用自动拉取，当达到拉取阈值**SHIFTCTRL_PULL_THRESH**时，OSR会自动从TX FIFO重新填充。输出移位计数同时被清零为0。在此情况下，如TX FIFO为空，**OUT**操作将暂停，否则仍在周期内执行。具体内容详见第3.5.4节。

OUT EXEC允许指令以内联形式包含在FIFO数据流中。该 **OUT**本身在一个周期内执行，而OSR中的指令则在下一个周期执行。此机制执行的指令类型无限制。初始 **OUT**指令的延迟周期将被忽略，但被执行指令可正常插入延迟周期。

OUT PC 行为相当于无条件跳转至从OSR移出的地址。

3.4.5.3 汇编语法

out <destination>, <bit_count>

其中：

<**destination**> 为上述指定的目的地之一。

<**bit_count**> 为一个值（参见第3.3.2节），指定移位的位数（有效范围为1至32位）

3.4.6. PUSH

3.4.6.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
推送	1	0	0	延迟/副控				0	IfF	块	0	0	0	0	0	0

3.4.6.2. 操作

将ISR内容作为单个32位字推入RX FIFO。将ISR清零。

- **IfFull**: 如果为1，除非总输入移位计数达到其阈值**SHIFTCTRL_PUSH_THRESH**（与自动推送相同；详见第3.5.4节），否则不执行任何操作。
- **Block**: 如果为1，当RX FIFO已满时暂停执行。

PUSH IFFULL有助于使程序更紧凑，类似于自动推送。当启用自动推送导致 **IN**指令在不适当时间阻塞时，此功能尤其有用，例如此时状态机正在断言某些外部控制信号。

The PIO assembler sets the `Block` bit by default. If the `Block` bit is not set, the `PUSH` does not stall on a full RX FIFO, instead continuing immediately to the next instruction. The FIFO state and contents are unchanged when this happens. The ISR is still cleared to all-zeroes, and the `FDEBUG_RXSTALL` flag is set (the same as a blocking `PUSH` or autopush to a full RX FIFO) to indicate data was lost.

3.4.6.3. Assembler Syntax

```
push ( iffull )
push ( iffull ) block
push ( iffull ) noblock
```

where:

<code>iffull</code>	Is equivalent to <code>IfFull == 1</code> above. i.e. the default if this is not specified is <code>IfFull == 0</code>
<code>block</code>	Is equivalent to <code>Block == 1</code> above. This is the default if neither <code>block</code> nor <code>noblock</code> are specified
<code>noblock</code>	Is equivalent to <code>Block == 0</code> above.

3.4.7. PULL

3.4.7.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<code>PULL</code>	1	0	0	Delay/side-set				1	IfE	Blk	0	0	0	0	0	0

3.4.7.2. Operation

Load a 32-bit word from the TX FIFO into the OSR.

- `IfEmpty`: If 1, do nothing unless the total output shift count has reached its threshold, `SHIFTCTRL_PULL_THRESH` (the same as for autopull; see [Section 3.5.4](#)).
- `Block`: If 1, stall if TX FIFO is empty. If 0, pulling from an empty FIFO copies scratch X to OSR.

Some peripherals (UART, SPI...) should halt when no data is available, and pick it up as it comes in; others (I2S) should clock continuously, and it is better to output placeholder or repeated data than to stop clocking. This can be achieved with the `Block` parameter.

A nonblocking `PULL` on an empty FIFO has the same effect as `MOV OSR, X`. The program can either preload scratch register X with a suitable default, or execute a `MOV X, OSR` after each `PULL NOBLOCK`, so that the last valid FIFO word will be recycled until new data is available.

`PULL IFEMPTY` is useful if an `OUT` with autopull would stall in an inappropriate location when the TX FIFO is empty. `IfEmpty` permits some of the same program simplifications as autopull – for example, the elimination of an outer loop counter – but the stall occurs at a controlled point in the program.

PIO汇编器默认设置 `Block`位。若未设置 `Block`位，`PUSH`指令在RX FIFO已满时不会阻塞，而是立即继续执行下一条指令。发生此情况时，FIFO的状态和内容保持不变。ISR仍被清零，且`FDEBUG_RXSTALL`标志被设置（与阻塞状态的`PUSH`或自动推送（到已满的RX FIFO）相同），以指示数据丢失。

3.4.6.3. 汇编语法

```
push ( iffull )
push ( iffull ) block
push ( iffull ) noblock
```

说明：

- `iffull` 等同于上述的`IfFull == 1`。即未指定时默认为`IfFull == 0`。
- `block` 等同于上述的`Block == 1`。若未指定`block`或`noblock`，则此项为默认。
- `noblock` 等同于上述的`Block == 0`。

3.4.7. PULL

3.4.7.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
拉取	1	0	0	延迟/副控				1	IfE	块	0	0	0	0	0	0

3.4.7.2. 操作

从TX FIFO加载一个32位字至OSR。

- `IfEmpty`：若为1，则除非总输出移位计数达到阈值`SHIFTCTRL_PULL_THRESH`（与`autopull`相同；详见第3.5.4节），否则不执行任何操作。
- `Block`：若为1，则当TX FIFO为空时暂停执行。若为0，从空FIFO中读取时，将寄存器X的内容复制到OSR。

某些外设（UART、SPI等）在无数据时应暂停，并在数据到达时继续处理；其他外设（I2S）应持续时钟运行，输出占位符或重复数据优于停止时钟。此功能可通过`Block`参数实现。

在空FIFO上执行非阻塞`PULL`，其效果等同于`MOV OSR, X`指令。程序可预先将寄存器X加载合适默认值，或在每次`PULL NOBLOCK`之后执行`MOV X, OSR`，以便在新数据到达前重复使用最后一条有效FIFO数据。

当TX FIFO为空且自动拉取（`autopull`）的`OUT`操作将在不适当的位置停顿时，`PULL IFEMPTY`指令非常有用。`IfEmpty`允许实现与自动拉取相同的程序简化，例如消除外层循环计数器，但停顿会发生在程序的受控位置。

NOTE

When autopull is enabled, any **PULL** instruction is a no-op when the OSR is full, so that the **PULL** instruction behaves as a barrier. **OUT NULL, 32** can be used to explicitly discard the OSR contents. See [Section 3.5.4.2](#) for more detail.

3.4.7.3. Assembler Syntax

```
pull ( ifempty )
pull ( ifempty ) block
pull ( ifempty ) noblock
```

where:

<i>ifempty</i>	Is equivalent to IfEmpty == 1 above. i.e. the default if this is not specified is IfEmpty == 0
<i>block</i>	Is equivalent to Block == 1 above. This is the default if neither <i>block</i> nor <i>noblock</i> are specified
<i>noblock</i>	Is equivalent to Block == 0 above.

3.4.8. MOV**3.4.8.1. Encoding**

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	Delay/side-set				Destination			Op		Source			

3.4.8.2. Operation

Copy data from **Source** to **Destination**.

- Destination:
 - 000: **PINS** (Uses same pin mapping as **OUT**)
 - 001: **X** (Scratch register X)
 - 010: **Y** (Scratch register Y)
 - 011: Reserved
 - 100: **EXEC** (Execute data as instruction)
 - 101: **PC**
 - 110: **ISR** (Input shift counter is reset to 0 by this operation, i.e. empty)
 - 111: **OSR** (Output shift counter is reset to 0 by this operation, i.e. full)
- Operation:
 - 00: None
 - 01: Invert (bitwise complement)
 - 10: Bit-reverse
 - 11: Reserved

ⓘ 注意

启用自动拉取 (autopull) 时，当OSR已满，任何 **PULL** 指令均为无操作，因此 **PULL** 指令表现为屏障。**OUT NULL, 32** 可用于显式丢弃OSR内容。详见第3.5.4.2节。

3.4.7.3 汇编器语法

pull (ifempty)

pull (ifempty) 块

pull (ifempty) 非块

说明：

ifempty 相当于前述的 **IfEmpty == 1**。即若未指定，默认条件为 **IfEmpty == 0**。

block 等同于上述的 **Block == 1**。若未指定 **block** 或 **noblock**，则此项为默认。

noblock 等同于上述的 **Block == 0**。

3.4.8. MOV

3.4.8.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	延迟/副控				目标			操作		来源			

3.4.8.2. 操作

从 **Source** 复制数据至 **Destination**。

• 目的地：

- 000: **PINS** (使用与 **OUT** 相同的引脚映射)
- 001: **X** (临时寄存器 X)
- 010: **Y** (临时寄存器 Y)
- 011: 保留
- 100: **EXEC** (将数据作为指令执行)
- 101: **PC**
- 110: **ISR** (该操作将输入移位计数器重置为 0，即清空)
- 111: **OSR** (该操作将输出移位计数器重置为 0，即填满)

• 操作：

- 00: 无
- 01: 反转 (按位取反)
- 10: 位反转
- 11: 保留

- Source:

- 000: PINS (Uses same pin mapping as IN)
- 001: X
- 010: Y
- 011: NULL
- 100: Reserved
- 101: STATUS
- 110: ISR
- 111: OSR

`MOV PC` causes an unconditional jump. `MOV EXEC` has the same behaviour as `OUT EXEC` (Section 3.4.5), and allows register contents to be executed as an instruction. The `MOV` itself executes in 1 cycle, and the instruction in `Source` on the next cycle. Delay cycles on `MOV EXEC` are ignored, but the executee may insert delay cycles as normal.

The `STATUS` source has a value of all-ones or all-zeroes, depending on some state machine status such as FIFO full/empty, configured by `EXECCTRL_STATUS_SEL`.

`MOV` can manipulate the transferred data in limited ways, specified by the `Operation` argument. Invert sets each bit in `Destination` to the logical NOT of the corresponding bit in `Source`, i.e. 1 bits become 0 bits, and vice versa. Bit reverse sets each bit n in `Destination` to bit $31 - n$ in `Source`, assuming the bits are numbered 0 to 31.

`MOV dst, PINS` reads pins using the `IN` pin mapping, and writes the full 32-bit value to the destination without masking. The LSB of the read value is the pin indicated by `PINCTRL_IN_BASE`, and each successive bit comes from a higher-numbered pin, wrapping after 31.

3.4.8.3. Assembler Syntax

`mov <destination>, (op) <source>`

where:

`<destination>` Is one of the destinations specified above.

`<op>` If present, is:

`!` or `~` for NOT (Note: this is always a bitwise NOT)

`::` for bit reverse

`<source>` Is one of the sources specified above.

3.4.9. IRQ

3.4.9.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	Delay/side-set				0	Clr	Wait	Index					

- 来源：

- 000: PINS (使用与 IN相同的引脚映射)
- 001: X
- 010: Y
- 011: NULL
- 100: 保留
- 101: 状态
- 110: ISR
- 111: OSR

MOV PC会导致无条件跳转。**MOV EXEC**的行为与**OUT EXEC**（第3.4.5节）相同，允许将寄存器内容作为指令执行。**MOV**本身执行1个周期，下一周期执行Source中的指令。**MOV EXEC**上的延迟周期会被忽略，但被执行指令可正常插入延迟周期。

STATUS源的值为全1或全0，取决于某些状态机状态（如FIFO已满/已空），由EXECCTRL_STATUS_SEL配置。

MOV可对传输的数据进行有限的操作，具体由 **Operation**参数指定。反转操作将目标寄存器Destination中的每个位设为对应源寄存器Source位的逻辑非，即1变0，0变1。位逆操作将目标寄存器Destination中的每个位n设为源寄存器Source中第31-n位的值，假定位编号为0至31。

MOV dst, PINS 使用 IN引脚映射读取引脚，并将完整的32位值写入目标寄存器，无需掩码。

读取值的最低有效位（LSB）对应PINCTRL_IN_BASE指定的引脚，后续的每个位来自编号更高的引脚，超过31时回绕。

3.4.8.3. 汇编语法

`mov <destination>, (op) <source>`

说明：

<destination> 为上述指定的目的地之一。

<op> 若存在，格式为：

! 或 ~表示逻辑非（注意：始终为按位非）

:: 表示位反转

<source> 为上述指定的某一源。

3.4.9. IRQ

3.4.9.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	延迟/副控				0	清除	等待	索引					

3.4.9.2. Operation

Set or clear the IRQ flag selected by `Index` argument.

- Clear: if 1, clear the flag selected by `Index`, instead of raising it. If `Clear` is set, the `Wait` bit has no effect.
- Wait: if 1, halt until the raised flag is lowered again, e.g. if a system interrupt handler has acknowledged the flag.
- Index:
 - The 3 LSBs specify an IRQ index from 0-7. This IRQ flag will be set/cleared depending on the Clear bit.
 - If the MSB is set, the state machine ID (0...3) is added to the IRQ index, by way of modulo-4 addition on the two LSBs. For example, state machine 2 with a flag value of 0x11 will raise flag 3, and a flag value of 0x13 will raise flag 1.

IRQ flags 4-7 are visible only to the state machines; IRQ flags 0-3 can be routed out to system level interrupts, on either of the PIO's two external interrupt request lines, configured by `IRQ0_INTE` and `IRQ1_INTE`.

The modulo addition bit allows relative addressing of 'IRQ' and 'WAIT' instructions, for synchronising state machines which are running the same program. Bit 2 (the third LSB) is unaffected by this addition.

If `Wait` is set, `Delay` cycles do not begin until after the wait period elapses.

3.4.9.3. Assembler Syntax

```
irq <irq_num> ( rel )
irq set <irq_num> ( rel )
irq nowait <irq_num> ( rel )
irq wait <irq_num> ( rel )
irq clear <irq_num> ( rel )
```

where:

<code><irq_num> (rel)</code>	Is a value (see Section 3.3.2) specifying The irq number to wait on (0-7). If <code>rel</code> is present, then the actual irq number used is calculating by replacing the low two bits of the irq number (irq_num_{10}) with the low two bits of the sum ($irq_num_{10} + sm_num_{10}$) where sm_num_{10} is the state machine number
<code>irq</code>	Means set the IRQ without waiting
<code>irq set</code>	Also means set the IRQ without waiting
<code>irq nowait</code>	Again, means set the IRQ without waiting
<code>irq wait</code>	Means set the IRQ and wait for it to be cleared before proceeding
<code>irq clear</code>	Means clear the IRQ

3.4.10. SET

3.4.10.1. Encoding

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SET	1	1	1	Delay/side-set				Destination				Data				

3.4.9.2. 操作

设置或清除由 `Index` 参数指定的IRQ标志。

- 清除：若为1，则清除由 `Index` 指定的标志，而非置位。如果 `Clear` 被设置，则 `Wait` 位无效。
- 等待：若为1，则暂停执行，直到被置位的标志被清除，例如系统中断处理程序已确认该标志。
- 索引：
 - 最低三位（3 LSBs）指定一个IRQ索引，范围为0至7。该IRQ标志将根据清除位设置或清除。
 - 如果最高位（MSB）被置位，则通过对最低两位（2 LSBs）进行模4加法，将状态机ID（0至3）加至IRQ索引。例如，状态机2且标志值为0x11时将触发标志3，标志值为0x13时将触发标志1。

IRQ标志4至7仅对状态机可见；IRQ标志0至3可路由至系统级中断，经过PIO的两个外部中断请求线路中的任一条，该线路由 `IRQ0_INTE` 和 `IRQ1_INTE` 配置。

模加位允许 ‘IRQ’ 与 ‘WAIT’ 指令的相对寻址，以同步运行相同程序的状态机。第2位（第三个最低有效位）不受该加法影响。

如果 `Wait` 被设置，`Delay` 周期将在等待期结束后开始。

3.4.9.3. 汇编语法

```
irq <irq_num> (rel)
irq set <irq_num> (rel)
irq nowait <irq_num> (rel)
irq wait <irq_num> (rel)
irq clear <irq_num> (rel)
```

说明：

`<irq_num> (rel)` 为一个值（详见第3.3.2节），指定等待的IRQ编号（0-7）。如果 `rel` 存在，则实际使用的 `irq` 号通过将 `irq` 号 (irq_num_{10}) 的低两位替换为和 ($irq_num_{10} + sm_num_{10}$) 低两位计算得出，其中 sm_n 为状态机编号。

数字

<code>irq</code>	表示设置IRQ且不等待
<code>irq set</code>	同样表示设置IRQ且不等待
<code>irq nowait</code>	再次表示设置IRQ且不等待
<code>irq wait</code>	表示设置IRQ并等待其被清除后继续
<code>irq clear</code>	表示清除IRQ

3.4.10. SET

3.4.10.1. 编码

位:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
设置	1	1	1	延迟/副控				目标			数据					

3.4.10.2. Operation

Write immediate value `Data` to `Destination`.

- Destination:
 - 000: `PINS`
 - 001: `X` (scratch register X) 5 LSBs are set to `Data`, all others cleared to 0.
 - 010: `Y` (scratch register Y) 5 LSBs are set to `Data`, all others cleared to 0.
 - 011: Reserved
 - 100: `PINDIRS`
 - 101: Reserved
 - 110: Reserved
 - 111: Reserved
- Data: 5-bit immediate value to drive to pins or register.

This can be used to assert control signals such as a clock or chip select, or to initialise loop counters. As `Data` is 5 bits in size, scratch registers can be `SET` to values from 0-31, which is sufficient for a 32-iteration loop.

The mapping of `SET` and `OUT` onto pins is configured independently. They may be mapped to distinct locations, for example if one pin is to be used as a clock signal, and another for data. They may also be overlapping ranges of pins: a UART transmitter might use `SET` to assert start and stop bits, and `OUT` instructions to shift out FIFO data to the same pins.

3.4.10.3. Assembler Syntax

`set <destination>, <value>`

where:

- | | |
|----------------------------------|--|
| <code><destination></code> | Is one of the destinations specified above. |
| <code><value></code> | The value (see Section 3.3.2) to set (valid range 0-31) |

3.5. Functional Details

3.5.1. Side-set

Side-set is a feature that allows state machines to change the level or direction of up to 5 pins, concurrently with the main execution of the instruction.

One example where this is necessary is a fast SPI interface: here a clock transition (toggling 1→0 or 0→1) must be simultaneous with a data transition, where a new data bit is shifted from the OSR to a GPIO. In this case an `OUT` with a side-set would achieve both of these at once.

This makes the timing of the interface more precise, reduces the overall program size (as a separate `SET` instruction is not needed to toggle the clock pin), and also increases the maximum frequency the SPI can run at.

Side-set also makes GPIO mapping much more flexible, as its mapping is independent from `SET`. The example I2C code allows SDA and SCL to be mapped to any two arbitrary pins, if clock stretching is disabled. Normally, SCL toggles to synchronise data transfer, and SDA contains the data bits being shifted out. However, some particular I2C sequences such as `Start` and `Stop` line conditions, need a fixed pattern to be driven on SDA as well as SCL. The mapping I2C uses to achieve this is:

3.4.10.2. 操作

将立即数 `Data` 写入至 `Destination`。

- 目的地：
 - 000: 引脚
 - 001: `X` (临时寄存器 X) 低 5 位设为 `Data`, 其他位清零。
 - 010: `Y` (临时寄存器 Y) 低 5 位设为 `Data`, 其他位清零。
 - 011: 保留
 - 100: `PINDIRS`
 - 101: 保留
 - 110: 保留
 - 111: 保留
- `Data`: 用于驱动引脚或寄存器的 5 位立即数。

可用于断言控制信号, 如时钟或芯片选择, 或初始化循环计数器。由于 `Data` 为 5 位, 临时寄存器可通过 `SET` 设为 0 至 31 的值, 足以支持 32 次迭代的循环。

`SET` 和 `OUT` 指令映射至引脚的方式是独立配置的。它们可映射至不同位置, 例如一个引脚用作时钟信号, 另一个用于数据。也可以是引脚的重叠范围: UART 发送器可能用 `SET` 断言起始和停止位, 使用 `OUT` 指令将 FIFO 数据移出至相同引脚。

3.4.10.3. 汇编语法

`set <destination>, <value>`

其中:

- | | |
|----------------------------------|---------------------------------|
| <code><destination></code> | 为上述指定的目的地之一。 |
| <code><value></code> | 要设置的值 (参见第3.3.2节), 有效范围为 0 至 31 |

3.5. 功能细节

3.5.1. 旁路设定

Side-set 功能允许状态机在指令主执行的同时, 同时改变最多 5 个引脚的电平或方向。

一个典型应用为高速 SPI 接口: 时钟信号的跳变 (从 1→0 或 0→1) 必须与数据的跳变同步, 即通过 OSR 向 GPIO 移位新的数据位。在此情况下, 带 side-set 的 `OUT` 指令能够同时完成这两项操作。

该功能提高了接口的时序精度, 减小了程序整体尺寸 (无需额外 `SET` 指令切换时钟引脚), 并提升了 SPI 的最高工作频率。

此外, Side-set 使 GPIO 映射更为灵活, 其映射方式独立于 `SET` 指令。示例 I2C 代码允许在禁用时钟拉伸的情况下, 将 SDA 和 SCL 映射到任意两个引脚。通常, SCL 信号用于切换以同步数据传输, 而 SDA 包含被移出的数据位。然而, 一些特定的 I2C 序列, 如 `Start` 和 `Stop` 线条件, 需要在 SDA 及 SCL 上驱动固定的信号模式。I2C 为实现此目的所采用的映射是:

- Side-set → SCL
- OUT → SDA
- SET → SDA

This lets the state machine serve the two use cases of data on SDA and clock on SCL, or fixed transitions on both SDA and SCL, while still allowing SDA and SCL to be mapped to any two GPIOs of choice.

The side-set data is encoded in the `Delay/side-set` field of each instruction. Any instruction can be combined with side-set, including instructions which write to the pins, such as `OUT PINS` or `SET PINS`. Side-set's pin mapping is independent from `OUT` and `SET` mappings, though it may overlap. If side-set and an `OUT` or `SET` write to the same pin simultaneously, the side-set data is used.

NOTE

If an instruction stalls, the side-set still takes effect immediately.

```

1 .program spi_tx_fast
2 .side_set 1
3
4 loop:
5     out pins, 1  side 0
6     jmp loop      side 1

```

The `spi_tx_fast` example shows two benefits of this: data and clock transitions can be more precisely co-aligned, and programs can be made faster overall, with an output of one bit per two system clock cycles in this case. Programs can also be made smaller.

There are four things to configure when using side-set:

1. The number of MSBs of the `Delay/side-set` field to use for side-set rather than delay. This is configured by `PINCTRL_SIDESET_COUNT`. If this is set to 5, delay cycles are not available. If set to 0, no side-set will take place.
2. Whether to use the most significant of these bits as an enable. Side-set takes place on instructions where the enable is high. If there is no enable bit, `every` instruction on that state machine will perform a side-set, if `SIDESET_COUNT` is nonzero. This is configured by `EXECCTRL_SIDE_EN`.
3. The GPIO number to map the least-significant side-set bit to. Configured by `PINCTRL_SIDESET_BASE`.
4. Whether side-set writes to GPIO levels or GPIO directions. Configured by `EXECCTRL_SIDE_PINDIR`

In the above example, we have only one side-set data bit, and every instruction performs a side-set, so no enable bit is required. `SIDESET_COUNT` would be 1, `SIDE_EN` would be false. `SIDE_PINDIR` would also be false, as we want to drive the clock high and low, not high- and low-impedance. `SIDESET_BASE` would select the GPIO the clock is driven from.

3.5.2. Program Wrapping

PIO programs often have an "outer loop": they perform the same sequence of steps, repetitively, as they transfer a stream of data between the FIFOs and the outside world. The square wave program from the introduction is a minimal example of this:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> Lines 8 - 13

```

8 .program squarewave
9     set pindirs, 1    ; Set pin to output
10 again:
11     set pins, 1 [1]   ; Drive pin high and then delay for one cycle
12     set pins, 0        ; Drive pin low

```

- Side-set → SCL
- OUT → SDA
- SET → SDA

此设计允许状态机同时满足两种使用场景：SDA上传输数据且SCL作为时钟，或在SDA与SCL上产生固定的切换信号，同时仍支持将SDA和SCL映射至任意两个所选GPIO。

side-set数据编码在每条指令的Delay/side-set字段中。任何指令均可与side-set组合，包括向引脚写入的指令，如 OUT PINS 或 SET PINS。边集的引脚映射独立于 OUT 和 SET 映射，尽管可能存在重叠。若边集与 OUT 或 SET 同时写入同一引脚，则以边集数据为准。

注意

即使指令阻塞，边集仍会立即生效。

```

1 .program spi_tx_fast
2 .side_set 1
3
4 loop:
5     输出引脚，1侧 0
6     jmp loop      边 1

```

spi_tx_fast示例表明两大优势：数据与时钟跳变可更精确对齐，且程序整体加速，当前情况下每两个系统时钟周期输出一位。程序也可变得更小。

使用边集时需配置四项参数：

1. 用于边集而非延迟的Delay/side-set字段最高有效位数。由PINCTRL_SIDESET_COUNT进行配置。若设置为5，则无法使用延迟周期。如果设置为0，则不会发生侧设定。
2. 是否使用这些位中的最高有效位作为启用位。只有当启用位为高时，侧设定才会在指令上执行。如果没有启用位，且SIDESET_COUNT非零，则该状态机上的每条指令都会执行侧设定。该配置由EXECCTRL_SIDE_EN进行设置。
3. 映射最低有效侧设定位的GPIO编号。由PINCTRL_SIDESET_BASE进行配置。
4. 侧设定是写入GPIO电平还是GPIO方向。由EXECCTRL_SIDE_PINDIR进行配置。

在上述示例中，只有一个侧设定数据位，且每条指令均执行侧设定，因此不需要启用位。SIDESET_COUNT设置为1，SIDE_EN设置为假。SIDE_PINDIR亦为假，因为我们希望驱动时钟的高低电平，而非高阻和低阻状态。SIDESET_BASE用于选择时钟所驱动的GPIO端口。

3.5.2. 程序封装

PIO程序通常包含“外层循环”：它们重复执行相同的步骤序列，在FIFO与外部世界之间传输数据流。引言中的方波程序即为此类最简示例：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave.pio> 第8至13行

```

8 .program squarewave
9     set pindirs, 1      ; 设置引脚为输出
10 again:
11     set pins, 1 [1] ; 将引脚置为高电平，随后延迟一个周期
12     set pins, 0          ; 将引脚置为低电平

```

```
13    jmp again           ; Set PC to label `again`
```

The main body of the program drives a pin high, and then low, producing one period of a square wave. The entire program then loops, driving a periodic output. The jump itself takes one cycle, as does each `set` instruction, so to keep the high and low periods of the same duration, the `set pins, 1` has a single delay cycle added, which makes the state machine idle for one cycle before executing the `set pins, 0` instruction. In total, each loop takes four cycles. There are two frustrations here:

- The `JMP` takes up space in the instruction memory that could be used for other programs
- The extra cycle taken to execute the `JMP` ends up *halving* the maximum output rate

As the Program Counter (`PC`) naturally wraps to 0 when incremented past 31, we could solve the second of these by filling the entire instruction memory with a repeating pattern of `set pins, 1` and `set pins, 0`, but this is wasteful. State machines have a hardware feature, configured via their `EXECCTRL` control register, which solves this common case.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_wrap.pio Lines 12 - 20

```
12 .program squarewave_wrap
13 ; Like squarewave, but use the state machine's .wrap hardware instead of an
14 ; explicit jmp. This is a free (0-cycle) unconditional jump.
15
16     set pindirs, 1    ; Set pin to output
17 .wrap_target
18     set pins, 1 [1]   ; Drive pin high and then delay for one cycle
19     set pins, 0 [1]   ; Drive pin low and then delay for one cycle
20 .wrap
```

After executing an instruction from the program memory, state machines use the following logic to update `PC`:

1. If the current instruction is a `JMP`, and the `Condition` is true, set `PC` to the `Target`
2. Otherwise, if `PC` matches `EXECCTRL_WRAP_TOP`, set `PC` to `EXECCTRL_WRAP_BOTTOM`
3. Otherwise, increment `PC`, or set to 0 if the current value is 31.

The `.wrap_target` and `.wrap` assembly directives in `pioasm` are essentially labels. They export constants which can be written to the `WRAP_BOTTOM` and `WRAP_TOP` control fields, respectively:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/generated/squarewave_wrap.pio.h

```
1 // -----
2 // This file is autogenerated by pioasm; do not edit! //
3 // -----
4
5 #pragma once
6
7 #include "hardware/pio.h"
8
9 // -----
10 // squarewave_wrap //
11 // -----
12
13 #define squarewave_wrap_target 1
14 #define squarewave_wrap_wrap 2
15 #define squarewave_wrap_pio_version 0
16
17 static const uint16_t squarewave_wrap_program_instructions[] = {
18     0xe081, // 0: set pindirs, 1
19                 // .wrap_target
20     0xe101, // 1: set pins, 1 [1]
21     0xe100, // 2: set pins, 0 [1]
```

```
13     jmp again           ; 将程序计数器设置为标签 `again`
```

程序主体将引脚拉高，然后拉低，产生一个完整的方波周期。整个程序循环执行，实现周期性输出。跳转指令本身占用一个周期，每条 `set` 指令亦是如此。为了保持高电平和低电平的时长一致，`set pins, 1` 指令增加了一个延迟周期，使状态机在执行 `set pins, 0` 指令前空闲一个周期。总体而言，每个循环耗时四个周期。此处存在两个不足之处：

- `JMP` 指令占用指令存储空间，减少了可用于其他程序的空间。
- 执行 `JMP` 指令所需的额外周期最终导致最大输出速率降低一半。

由于程序计数器（`PC`）在超过31后自然回绕至0，我们可以通过将整个指令存储器填充为重复的 `set pins, 1` 与 `set pins, 0` 模式来解决第二个问题，但这存在资源浪费。状态机具备通过其 `EXECCTRL` 控制寄存器配置的硬件特性，能够解决此类常见情况。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_wrap.pio 第12至20行

```
12 .program squarewave_wrap
13 ; 类似于squarewave，但使用状态机的.wrap硬件功能替代
14 ; 显式jmp。这是一个免费的（0周期）无条件跳转指令。
15
16     set pindirs, 1      ; 将引脚设置为输出
17 .wrap_target
18     set pins, 1 [1] ; 将引脚置为高电平，随后延迟一个周期
19     set pins, 0 [1] ; 将引脚拉低，然后延迟一个周期
20 .wrap
```

从程序存储器执行指令后，状态机使用以下逻辑更新 `PC`：

1. 如果当前指令为 `JMP`，且 `Condition` 为真，则将 `PC` 设置为 `Target`
2. 否则，若 `PC` 匹配 `EXECCTRL_WRAP_TOP`，则将 `PC` 设置为 `EXECCTRL_WRAP_BOTTOM`
3. 否则，递增 `PC`；若当前值为31，则将其设置为0。

`.wrap_target` 和 `.wrap` 汇编指令在 `pioasm` 中实质上为标签。它们导出可分别写入 `WRAP_BOTTOM` 和 `WRAP_TOP` 控制字段的常量：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/generated/squarewave_wrap.pio.h

```
1 // -----
2 // 本文件由 pioasm 自动生成，禁止编辑！ //
3 // -----
4
5 #pragma once
6
7 #include "hardware/pio.h"
8
9 // -----
10 // squarewave_wrap //
11 // -----
12
13 #define squarewave_wrap_target 1
14 #define squarewave_wrap_wrap 2
15 #define squarewave_wrap_pio_version 0
16
17 static const uint16_t squarewave_wrap_program_instructions[] = {
18     0xe081, // 0: 设置 pindirs, 1
19                 //     .wrap_target
20     0xe101, // 1: 设置 pins, 1
21     0xe100, // 2: 设置 pins, 0
```

```

22         //      .wrap
23 };
24
25 static const struct pio_program squarewave_wrap_program = {
26     .instructions = squarewave_wrap_program_instructions,
27     .length = 3,
28     .origin = -1,
29     .pio_version = squarewave_wrap_pio_version,
30     .used_gpio_ranges = 0x0
31 #endif
32 };
33
34 static inline pio_sm_config squarewave_wrap_program_get_default_config(uint offset) {
35     pio_sm_config c = pio_get_default_sm_config();
36     sm_config_set_wrap(&c, offset + squarewave_wrap_target, offset +
37                         squarewave_wrap_wrap);
38 }

```

This is raw output from the PIO assembler, `pioasm`, which has created a default `pio_sm_config` object containing the `WRAP` register values from the program listing. The control register fields could also be initialised directly.

NOTE

`WRAP_BOTTOM` and `WRAP_TOP` are absolute addresses in the PIO instruction memory. If a program is loaded at an offset, the wrap addresses must be adjusted accordingly.

The `squarewave_wrap` example has delay cycles inserted, so that it behaves identically to the original `squarewave` program. Thanks to program wrapping, these can now be removed, so that the output toggles twice as fast, while maintaining an even balance of high and low periods.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_fast.pio Lines 12 - 18

```

12 .program squarewave_fast
13 ; Like squarewave_wrap, but remove the delay cycles so we can run twice as fast.
14     set pindirs, 1    ; Set pin to output
15 .wrap_target
16     set pins, 1        ; Drive pin high
17     set pins, 0        ; Drive pin low
18 .wrap

```

3.5.3. FIFO Joining

By default, each state machine possesses a 4-entry FIFO in each direction: one for data transfer from system to state machine (TX), the other for the reverse direction (RX). However, many applications do not require bidirectional data transfer between the system and an individual state machine, but may benefit from deeper FIFOs: in particular, high-bandwidth interfaces such as DPI. For these cases, `SHIFTCTRL_FJOIN` can merge the two 4-entry FIFOs into a single 8-entry FIFO.

```

22         //      .wrap
23 };
24
25 static const struct pio_program squarewave_wrap_program = {
26     .instructions = squarewave_wrap_program_instructions,
27     .length = 3,
28     .origin = -1,
29     .pio_version = squarewave_wrap_pio_version,
30     .used_gpio_ranges = 0x0
31 #endif
32 };
33
34 static inline pio_sm_config squarewave_wrap_program_get_default_config(uint offset) {
35     pio_sm_config c = pio_get_default_sm_config();
36     sm_config_set_wrap(&c, offset + squarewave_wrap_target, offset +
37                         squarewave_wrap_wrap);
38 }

```

这是来自PIO汇编器 `pioasm` 的原始输出，生成了一个默认的 `pio_sm_config` 对象，其中包含程序列表中的 `WRAP` 寄存器值。控制寄存器字段也可以直接初始化。

注意

`WRAP_BOTTOM` 和 `WRAP_TOP` 是 PIO 指令存储器中的绝对地址。如果程序以偏移量加载，则必须相应调整包裹地址。

`squarewave_wrap` 示例插入了延迟周期，因此其行为与原始的 `quarewave` 程序完全一致。得益于程序包裹功能，这些延迟现在可以移除，从而使输出的切换速度加倍，同时保持高低周期的均衡。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/squarewave/squarewave_fast.pio 第12 - 18行

```

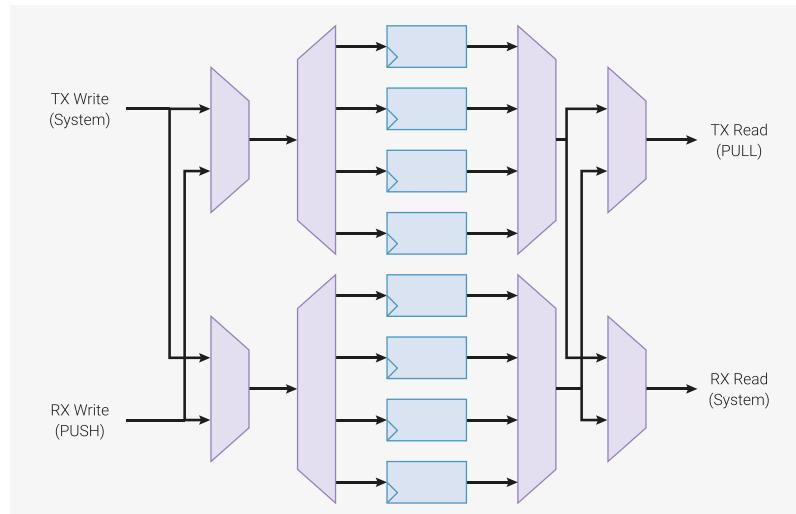
12 .program squarewave_fast
13 ; 类似于 squarewave_wrap，但去除延时周期，因此运行速度加倍。
14     set pindirs, 1      ; 将引脚设置为输出
15 .wrap_target
16     set pins, 1          ; 将引脚置高
17     set pins, 0          ; 将引脚置为低电平
18 .wrap

```

3.5.3. FIFO 连接

默认情况下，每个状态机在每个方向拥有一个4条目FIFO：一条用于系统向状态机传输数据（TX），另一条用于相反方向（RX）。然而，许多应用不需要系统与单个状态机之间的双向数据传输，但可能需要更深的FIFO，特别是在高带宽接口如DPI中。对于这些情况，`SHIFTCTRL_FJOIN` 可将两个4条目FIFO合并为单个8条目FIFO。

Figure 42. Joinable dual FIFO. A pair of four-entry FIFOs, implemented with four data registers, a 1:4 decoder and a 4:1 multiplexer. Additional multiplexing allows write data and read data to cross between the TX and RX lanes, so that all 8 entries are accessible from both ports



Another example is a UART: because the TX/CTS and RX/RTS parts of a UART are asynchronous, they are implemented on two separate state machines. It would be wasteful to leave half of each state machine's FIFO resources idle. The ability to join the two halves into just a TX FIFO for the TX/CTS state machine, or just an RX FIFO in the case of the RX/RTS state machine, allows full utilisation. A UART equipped with an 8-deep FIFO can be left alone for twice as long between interrupts as one with only a 4-deep FIFO.

When one FIFO is increased in size (from 4 to 8), the other FIFO on that state machine is reduced to zero. For example, if joining to TX, the RX FIFO is unavailable, and any **PUSH** instruction will stall. The RX FIFO will appear both **RXFULL** and **RXEMPTY** in the **FSTAT** register. The converse is true if joining to RX: the TX FIFO is unavailable, and the **TXFULL** and **TXEMPTY** bits for this state machine will both be set in **FSTAT**. Setting both **FJOIN_RX** and **FJOIN_TX** makes both FIFOs unavailable.

8 FIFO entries is sufficient for 1 word per clock through the RP2040 system DMA, provided the DMA is not slowed by contention with other masters.

⚠ CAUTION

Changing **FJOIN** discards any data present in the state machine's FIFOs. If this data is irreplaceable, it must be drained beforehand.

3.5.4. Autopush and Autopull

With each **OUT** instruction, the OSR gradually empties, as data is shifted out. Once empty, it must be refilled: for example, a **PULL** transfers one word of data from the TX FIFO to the OSR. Similarly, the ISR must be emptied once full. One approach to this is a loop which performs a **PULL** after an appropriate amount of data has been shifted:

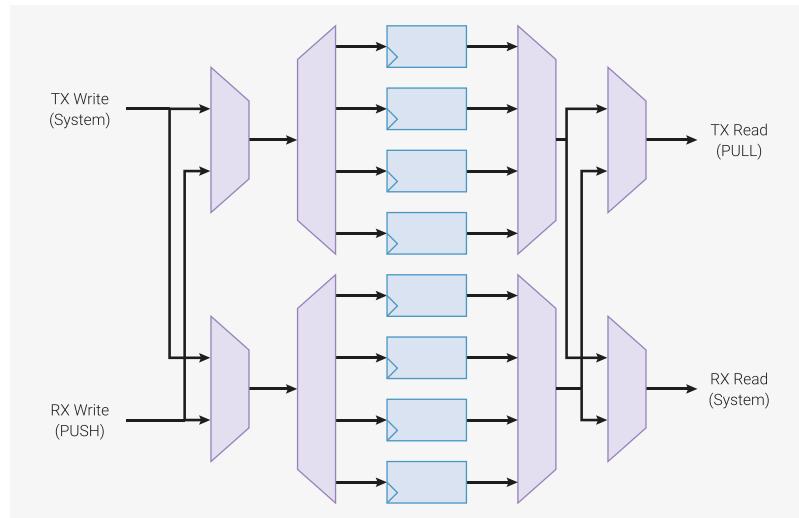
```

1 .program manual_pull
2 .side_set 1 opt
3
4 .wrap_target
5   set x, 2           ; X = bit count - 2
6   pull      side 1 [1] ; Stall here if no TX data
7 bitloop:
8   out pins, 1    side 0 [1] ; Shift out data bit and toggle clock low
9   jmp x-- bitloop side 1 [1] ; Loop runs 3 times
10  out pins, 1   side 0      ; Shift out last bit before reloading X
11 .wrap

```

This program shifts out 4 bits from each FIFO word, with an accompanying bit clock, at a constant rate of 1 bit per 4 cycles. When the TX FIFO is empty, it stalls with the clock high (noting that side-set still takes place on cycles where the

图42. 可合并的双FIFO。由四个数据寄存器、一个1:4译码器及一个4:1多路复用器组成的一对四条目FIFO。额外的复用允许写数据和读数据在TX和RX通道之间交叉传输，使所有8个条目均可从两个端口访问。



另一个例子是UART：由于UART的TX/CTS和RX/RTS部分是异步的，故实现于两个独立的状态机。让每个状态机一半的FIFO资源闲置属于极大浪费。将两半合并为仅供TX/CTS状态机使用的TX FIFO，或仅供RX/RTS状态机使用的RX FIFO，实现了资源的充分利用。配备8级FIFO的UART可使中断间隔时间比仅具4级FIFO的延长一倍。

当一个FIFO大小增加（由4增至8）时，同一状态机上的另一个FIFO会被减少至零。例如，如果合并至TX，则RX FIFO不可用，任何 **PUSH** 指令将被阻塞。RX FIFO将在 FSTAT 寄存器中同时显示为 RXFULL 和 RXEMPTY。如果连接至RX，则相反情况成立：TX FIFO不可用，且该状态机的 TXFULL 和 TXEMPTY 位均在 FSTAT 中被置位。同时设置 FJOIN_RX 和 FJOIN_TX 将导致两个FIFO均不可用。

8个FIFO项对于通过RP2040系统DMA每时钟周期传输1个字是足够的，前提是DMA未因与其他主控器竞争而减速。

⚠ 注意

更改 **FJOIN** 会丢弃状态机 FIFO 中存在的所有数据。如该数据不可替代，必须先行清空。

3.5.4. 自动推送与自动拉取

每执行一个 **OUT** 指令，OSR会随着数据移出而逐渐清空。OSR清空后必须重新填充：例如，使用 **PULL** 将一个字的数据从 TX FIFO 传输至 OSR。同理，ISR在填满后也必须清空。一种方法是在移动适量数据后通过循环执行 **PULL** 操作，如下所示：

```

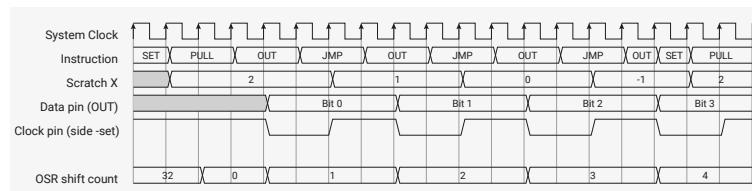
1 .program manual_pull
2 .side_set 1 opt
3
4 .wrap_target
5   set x, 2           ; X = 位数 - 2
6   pull             side 1 [1] ; 如果无TX数据，则在此处停顿
7 bitloop:
8   out pins, 1      side 0 [1] ; 移位输出数据位并将时钟拉低
9   jmp x-- bitloop side 1 [1] ; 循环执行3次
10  out pins, 1     side 0       ; 在重新加载X之前移位输出最后一位
11 .wrap

```

此程序以恒定速率（每4个周期1位）从每个FIFO字输出4位数据，并伴有位时钟。当TX FIFO为空时，程序将在时钟保持高电平状态下停顿（侧边设置side-set仍会在指令停顿周期执行）。

instruction stalls). Figure 43 shows how a state machine would execute this program.

Figure 43. Execution of manual_pull program. X is used as a loop counter. On each iteration, one data bit is shifted out, and the clock is asserted low, then high. A delay cycle on each instruction brings the total up to four cycles per iteration. After the third loop, a fourth bit is shifted out, and the state machine immediately returns to the start of the program to reload the loop counter and pull fresh data, while maintaining the 4 cycles/bit cadence.



This program has some limitations:

- It occupies 5 instruction slots, but only 2 of these are immediately useful (`out pins, 1 set 0` and `... set 1`), for outputting serial data and a clock.
- Its throughput is limited to system clock over 4, due to the extra cycles required to pull in new data, and reload the loop counter

This is a common type of problem for PIO, so each state machine has some extra hardware to handle it. State machines keep track of the total shift count `OUT` of the OSR and `IN` to the ISR, and trigger certain actions once these counters reach a programmable threshold.

- On an `OUT` instruction which reaches or exceeds the pull threshold, the state machine can simultaneously refill the OSR from the TX FIFO, if data is available.
- On an `IN` instruction which reaches or exceeds the push threshold, the state machine can write the shift result directly to the RX FIFO, and clear the ISR.

The `manual_pull` example can be rewritten to take advantage of automatic pull (autopull):

```

1 .program autopull
2 .side_set 1
3
4 .wrap_target
5   out pins, 1  side 0    [1]
6   nop          side 1    [1]
7 .wrap

```

This is shorter and simpler than the original, and can run twice as fast, if the delay cycles are removed, since the hardware refills the OSR "for free". Note that the program does not determine the total number of bits to be shifted before the next pull; the hardware automatically pulls once the programmable threshold, `SHIFCTRL_PULL_THRESH`, is reached, so the same program could also shift out e.g. 16 or 32 bits from each FIFO word.

Finally, note that the above program is not exactly the same as the original, since it stalls with the clock output low, rather than high. We can change the location of the stall, using the `PULL_IFEMPTY` instruction, which uses the same configurable threshold as autopull:

```

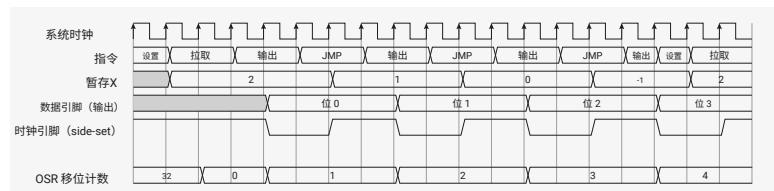
1 .program somewhat_manual_pull
2 .side_set 1
3
4 .wrap_target
5   out pins, 1  side 0    [1]
6   pull ifempty  side 1    [1]
7 .wrap

```

Below is a complete example (PIO program, plus a C program to load and run it) which illustrates autopull and autopush both enabled on the same state machine. It programs state machine 0 to loopback data from the TX FIFO to the RX FIFO, with a throughput of one word per two clocks. It also demonstrates how the state machine will stall if it tries to `OUT` when both the OSR and TX FIFO are empty.

图43展示了状态机如何执行该程序。

图 43。 manual_pul
I 程序的执行。
。 X 用作循环计数器。每次迭代移出一个数据位，时钟先被拉低，随后拉高。每条指令中包含一个延迟周期，使得每次迭代总周期数达到四个。第三次循环后，第四个位被移出，状态机立即返回。程序起始，重新加载循环计数器并拉取新数据，同时保持每位4周期的节奏。



该程序存在以下限制：

- 它占用5个指令槽，但仅2个指令立即有效（out pins, 1条设置为0，另一条设置为1），用于输出串行数据与时钟。
 - 由于拉取新数据和重新加载循环计数器需额外周期，其吞吐量限制为系统时钟频率的四分之一。
- 这是PIO中一种常见的问题类型，因此每个状态机配备了一些额外硬件来进行处理。状态机会跟踪OSR的总移位计数 OUT 和ISR的总移位计数 IN，并在这些计数达到可编程阈值时触发相应操作。
- 当 OUT 指令达到或超过拉取阈值时，若数据可用，状态机可同时从TX FIFO重新填充OSR。
 - 当 IN 指令达到或超过推送阈值时，状态机可将移位结果直接写入RX FIFO，且清除ISR。

manual_pull示例可重写以利用自动拉取（autopull）：

```
1 .program autopull
2 .side_set 1
3
4 .wrap_target
5     out pins, 1    side 0      [1]
6     nop            side 1      [1]
7 .wrap
```

该程序较原始版本更短更简洁，若去除延迟周期，运行速度可提升两倍，因硬件“免费”完成OSR的重新填充。请注意，该程序不会在下一次拉取之前确定总的移位位数；硬件在达到可编程阈值SHIFCTRL_PULL_THRESH后会自动拉取，因此相同程序亦可从每个FIFO字中移出例如16或32位。

最后，请注意，上述程序并非完全等同于原始程序，因为它在时钟输出为低电平时停滞，而非高电平状态。我们可以使用 PULL_IFEMPTY 指令改变停滞位置，该指令采用与自动拉取相同的可配置阈值：

```
1 .program somewhat_manual_pull
2 .side_set 1
3
4 .wrap_target
5     out pins, 1    side 0      [1]
6     pull_ifempty   side 1      [1]
7 .wrap
```

以下是一个完整示例（PIO程序及用于加载和运行该程序的C语言代码），演示在同一个状态机上同时启用自动拉取和自动推送。该程序配置状态机0，将数据从TX FIFO回环至RX FIFO，吞吐率为每两个时钟周期传输一个字。该示例还演示了当OSR和TX FIFO均为空时，状态机尝试执行 OUT 操作会导致停滞的情况。

```

1 .program auto_push_pull
2
3 .wrap_target
4     out x, 32
5     in x, 32
6 .wrap

```

```

1 #include "tb.h" // TODO this is built against existing sw tree, so that we get printf etc
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "auto_push_pull.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     // Load program and configure state machine 0 for autopush/pull with
15     // threshold of 32, and wrapping on program boundary. A threshold of 32 is
16     // encoded by a register value of 00000.
17     for (int i = 0; i < count_of(auto_push_pull_program); ++i)
18         mm_pio->instr_mem[i] = auto_push_pull_program[i];
19     mm_pio->sm[0].shiftctrl =
20         (1u << PIO_SM0_SHIFTCTRL_AUTOPUSH_LSB) |
21         (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB) |
22         (0u << PIO_SM0_SHIFTCTRL_PUSH_THRESH_LSB) |
23         (0u << PIO_SM0_SHIFTCTRL_PULL_THRESH_LSB);
24     mm_pio->sm[0].execctrl =
25         (auto_push_pull_wrap_target << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB) |
26         (auto_push_pull_wrap << PIO_SM0_EXECCTRL_WRAP_TOP_LSB);
27
28     // Start state machine 0
29     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
30
31     // Push data into TX FIFO, and pop from RX FIFO
32     for (int i = 0; i < 5; ++i)
33         mm_pio->txf[0] = i;
34     for (int i = 0; i < 5; ++i)
35         printf("%d\n", mm_pio->rx[0]);
36
37     return 0;
38 }

```

Figure 44 shows how the state machine executes the example program. Initially the OSR is empty, so the state machine stalls on the first OUT instruction. Once data is available in the TX FIFO, the state machine transfers this into the OSR. On the next cycle, the OUT can execute using the data in the OSR (in this case, transferring this data to the X scratch register), and the state machine simultaneously refills the OSR with fresh data from the FIFO. Since every IN instruction immediately fills the ISR, the ISR remains empty, and IN transfers data directly from scratch X to the RX FIFO.

```

1 .program auto_push_pull
2
3 .wrap_target
4     out x, 32
5     in x, 32
6 .wrap

```

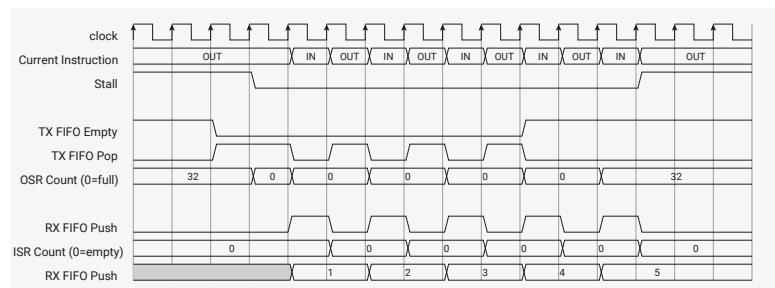
```

1 #include "tb.h" // TODO 本文件基于现有软件架构构建，以支持 printf 等功能
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "auto_push_pull.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     // 加载程序并配置状态机 0，启用自动推送/拉取
15     // 阈值设为 32，并在程序边界处自动回绕。阈值 32
16     // 由寄存器值 00000 编码。
17     for (int i = 0; i < count_of(auto_push_pull_program); ++i)
18         mm_pio->instr_mem[i] = auto_push_pull_program[i];
19     mm_pio->sm[0].shiftctrl =
20         (1u << PIO_SM0_SHIFTCTRL_AUTOPUSH_LSB) |
21         (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB) |
22         (0u << PIO_SM0_SHIFTCTRL_PUSH_THRESH_LSB) |
23         (0u << PIO_SM0_SHIFTCTRL_PULL_THRESH_LSB);
24     mm_pio->sm[0].execctrl =
25         (auto_push_pull_wrap_target << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB) |
26         (auto_push_pull_wrap << PIO_SM0_EXECCTRL_WRAP_TOP_LSB);
27
28     // 启动状态机 0
29     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
30
31     // 向 TX FIFO 推送数据，从 RX FIFO 弹出数据
32     for (int i = 0; i < 5; ++i)
33         mm_pio->txf[0] = i;
34     for (int i = 0; i < 5; ++i)
35         printf("%d\n", mm_pio->rx[0]);
36
37     return 0;
38 }

```

图44展示了状态机如何执行示例程序。最初OSR为空，因此状态机在第一个 OUT指令处暂停。一旦TX FIFO中有数据，状态机会将其传输到OSR。在下一周期， OUT可使用OSR中的数据执行（此处为将数据传输至X暂存寄存器），状态机同时从FIFO重新填充OSR。由于每条 IN指令会立即填充ISR，ISR保持为空，且 IN指令直接将数据从X暂存寄存器传输到RX FIFO。

Figure 44. Execution of auto_push_pull program. The state machine stalls on an OUT until data has travelled through the TX FIFO into the OSR. Subsequently, the OSR is refilled simultaneously with each OUT operation (due to bit count of 32), and IN data bypasses the ISR and goes straight to the RX FIFO. The state machine stalls again when the FIFO has drained, and the OSR is once again empty.



To trigger automatic push or pull at the correct time, the state machine tracks the total shift count of the ISR and OSR, using a pair of saturating 6-bit counters.

- At reset, or upon `CTRL_SM_RESTART` assertion, ISR shift counter is set to 0 (nothing shifted in), and OSR to 32 (nothing left to be shifted out)
- An `OUT` instruction increases the OSR shift counter by `Bit count`
- An `IN` instruction increases the ISR shift counter by `Bit count`
- A `PULL` instruction or autopull clears the OSR counter to 0
- A `PUSH` instruction or autopush clears the ISR counter to 0
- A `MOV OSR, x` or `MOV ISR, x` clears the OSR or ISR shift counter to 0, respectively
- A `OUT ISR, n` instruction sets the ISR shift counter to `n`

On any `OUT` or `IN` instruction, the state machine compares the shift counters to the values of `SHIFTCTRL_PULL_THRESH` and `SHIFTCTRL_PUSH_THRESH` to decide whether action is required. Autopull and autopush are individually enabled by the `SHIFTCTRL_AUTOPULL` and `SHIFTCTRL_AUTOPUSH` fields.

3.5.4.1. Autopush Details

Pseudocode for an 'IN' with autopush enabled:

```

1 isr = shift_in(isr, input())
2 isr count = saturate(isr count + in count)
3
4 if rx count >= threshold:
5     if rx fifo is full:
6         stall
7     else:
8         push(isr)
9     isr = 0
10    isr count = 0

```

Note that the hardware performs the above steps in a single machine clock cycle (unless there is a stall).

Threshold is configurable from 1 to 32.

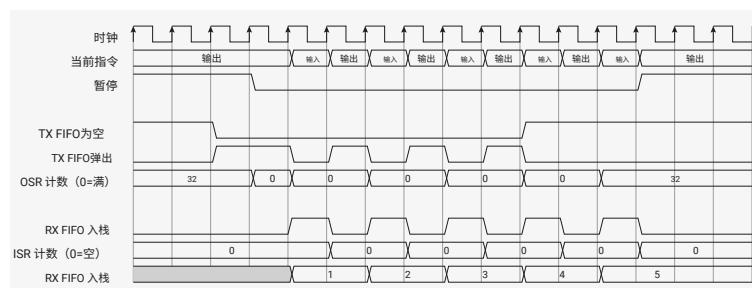
3.5.4.2. Autopull Details

On non-'OUT' cycles, the hardware performs the equivalent of the following pseudocode:

图 44。auto_push_pull 程序的执行。
状态机在 OUT 指令处暂停，直到数据通过 TX FIFO 传输至 OSR。

随后，由于位计数为 32，
OSR 在每次 OUT 操作时同步重新填充，且 IN 数据绕过 ISR 直接

进入 RX FIFO。当 FIFO 被清空时，状态机再次暂停。
OSR 也再次为空。



为了在正确时间触发自动推送或拉取，状态机使用一对饱和 6 位计数器跟踪 ISR 和 OSR 的总移位计数。

- 复位时，或在 `CTRL_SM_RESTART` 断言时，ISR 移位计数器设为 0（无移入数据），OSR 设为 32（无剩余待移出数据）
-
- 一条 OUT 指令将 OSR 移位计数器增加位计数。
- 一条 IN 指令使 ISR 移位计数器增加比特数
- 一条 PULL 指令或自动拉取将 OSR 计数器清零为 0
- 一条 PUSH 指令或自动推送将 ISR 计数器清零为 0
- 一条 `MOV OSR, n` 或 `MOV ISR, n` 指令分别将 OSR 或 ISR 移位计数器清零为 0
- 一条 `OUT ISR, n` 指令将 ISR 移位计数器设置为 n

在任何 OUT 或 IN 指令中，状态机将移位计数器与 `SHIFTCTRL_PULL_THRESH` 和 `SHIFTCTRL_PUSH_THRESH` 的值进行比较，以决定是否需要执行相应操作。自动拉取与自动推送分别通过 `SHIFTCTRL_AUTOPULL` 和 `SHIFTCTRL_AUTOPUSH` 字段启用。

3.5.4.1. 自动推送详细说明

启用自动推送功能的“IN”指令伪代码：

```

1 isr = shift_in(isr, input())
2 isr count = saturate(isr count + in count)
3
4 如果 rx 计数 ≥ 阈值：
5   如果 rx FIFO 已满：
6     暂停
7   else:
8     推送(isr)
9   isr = 0
10 _isr 计数 = 0

```

请注意，该硬件在单个机器时钟周期内完成上述步骤（除非发生暂停）。

阈值可配置，范围为 1 至 32。

3.5.4.2. 自动拉取详细信息

在非“OUT”周期内，硬件执行的操作等同于以下伪代码：

```

1 if MOV or PULL:
2     osr count = 0
3
4 if osr count >= threshold:
5     if tx fifo not empty:
6         osr = pull()
7         osr count = 0

```

An autopull can therefore occur at any point between two 'OUT' s, depending on when the data arrives in the FIFO.

On 'OUT' cycles, the sequence is a little different:

```

1 if osr count >= threshold:
2     if tx fifo not empty:
3         osr = pull()
4         osr count = 0
5     stall
6 else:
7     output(osr)
8     osr = shift(osr, out count)
9     osr count = saturate(osr count + out count)
10
11    if osr count >= threshold:
12        if tx fifo not empty:
13            osr = pull()
14            osr count = 0

```

The hardware is capable of refilling the OSR simultaneously with shifting out the last of the shift data, as these two operations can proceed in parallel. However, it cannot fill an empty OSR and 'OUT' it on the same cycle, due to the long logic path this would create.

The refill is somewhat asynchronous to your program, but an 'OUT' behaves as a data fence, and the state machine will never 'OUT' data which you didn't write into the FIFO.

Note that a 'MOV' from the OSR is undefined whilst autopull is enabled; you will read either any residual data that has not been shifted out, or a fresh word from the FIFO, depending on a race against system DMA. Likewise, a 'MOV' to the OSR may overwrite data which has just been autopulled. However, data which you 'MOV' into the OSR will never be overwritten, since 'MOV' updates the shift counter.

If you **do** need to read the OSR contents, you should perform an explicit 'PULL' of some kind. The nondeterminism described above is the cost of the hardware managing pulls automatically. When autopull is enabled, the behaviour of 'PULL' is altered: it becomes a no-op if the OSR is full. This is to avoid a race condition against the system DMA. It behaves as a fence: either an autopull has already taken place, in which case the 'PULL' has no effect, or the program will stall on the 'PULL' until data becomes available in the FIFO.

'PUSH' does not need a similar behaviour, because autopush does not have the same nondeterminism.

3.5.5. Clock Dividers

PIO runs off the system clock, but this is simply too fast for many interfaces, and the number of **Delay** cycles which can be inserted is limited. Some devices, such as UART, require the signalling rate to be precisely controlled and varied, and ideally multiple state machines can be varied independently while running identical programs. Each state machine is equipped with a clock divider, for this purpose.

Rather than slowing the system clock itself, the clock divider redefines how many system clock periods are considered to be "one cycle", for execution purposes. It does this by generating a clock enable signal, which can pause and resume execution on a per-system-clock-cycle basis. The clock divider generates clock enable pulses at regular intervals, so

```

1 如果 MOV 或 PULL:
2     osr 计数 = 0
3
4 如果 osr 计数 ≥ 阈值:
5     如果 tx FIFO 非空:
6         osr = pull()
7         osr 计数 = 0

```

因此，自动拉取可在两个“OUT”周期之间的任意时刻发生，具体取决于数据到达 FIFO 的时间。

在“OUT”周期内，执行顺序略有不同：

```

1 如果 osr 计数 ≥ 阈值:
2     如果 tx FIFO 非空:
3         osr = pull()
4         osr 计数 = 0
5     暂停
6 else:
7     output(osr)
8     osr = shift(osr, out count)
9     osr count = saturate(osr count + out count)
10
11    if osr count >= threshold:
12        如果 tx FIFO 非空:
13            osr = pull()
14            osr 计数 = 0

```

硬件能够在移出最后一部分移位数据的同时重新填充OSR，因为这两个操作可以并行执行。然而，不能在同一周期内填充空的OSR并执行“OUT”操作，因这将导致较长的逻辑路径。

重新填充与您的程序具有一定的异步性，但“OUT”操作表现为数据屏障，状态机将永远不会“OUT”未写入FIFO的数据。

请注意，当启用autopull时，从OSR执行的“MOV”操作行为未定义；您可能读取到尚未移出残留的数据，或来自FIFO的新字，这取决于与系统DMA的竞争状态。同样，向OSR执行的“MOV”操作可能覆盖刚自动拉取的数据。然而，您通过‘MOV’指令写入OSR的数据将永远不会被覆盖，因为‘MOV’会更新移位计数器。

如果您确实需要读取OSR内容，应执行显式的‘PULL’操作。上述非确定性是硬件自动管理拉取操作所带来的代价。启用自动拉取（autopull）后，‘PULL’指令的行为将改变：如果OSR已满，则该指令不执行任何操作。此设计旨在避免与系统DMA发生竞态条件。其行为类似屏障操作：要么自动拉取已完成，此时‘PULL’无效；要么程序将在‘PULL’指令处阻塞，直至FIFO中出现可用数据。

‘PUSH’无需类似行为，因为自动推送（autopush）不存在相同的非确定性。

3.5.5. 时钟分频器

PIO以系统时钟为基础运行，但对于许多接口而言速度过快，可插入的Delay周期数量有限。某些设备，例如UART，要求精确控制并变化信号速率，理想情况下，多个状态机可在运行相同程序的同时独立变化。为此，每个状态机均配备有时钟分频器。

时钟分频器不是直接降低系统时钟本身，而是重新定义执行时“一个周期”包含的系统时钟周期数。其通过生成时钟使能信号实现，该信号可在每个系统时钟周期内暂停或恢复执行。时钟分频器以固定间隔生成时钟使能脉冲，

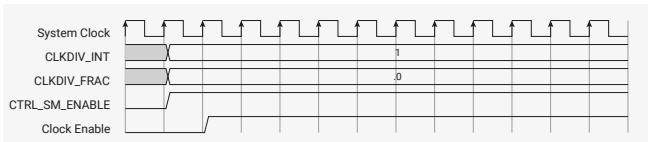
that the state machine runs at some steady pace, potentially much slower than the system clock.

Implementing the clock dividers in this way allows interfacing between the state machines and the system to be simpler, lower-latency, and with a smaller footprint. The state machine is completely idle on cycles where clock enable is low, though the system can still access the state machine's FIFOs and change its configuration.

The clock dividers are 16-bit integer, 8-bit fractional, with first-order delta-sigma for the fractional divider. The clock divisor can vary between 1 and 65536, in increments of $1 / 256$.

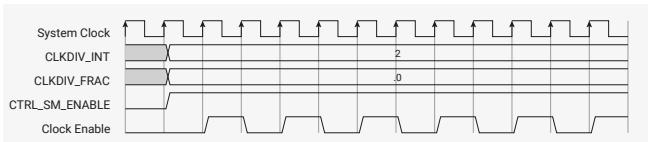
If the clock divisor is set to 1, the state machine runs on every cycle, i.e. full speed:

Figure 45. State machine operation with a clock divisor of 1. Once the state machine is enabled via the `CTRL` register, its clock enable is asserted on every cycle.



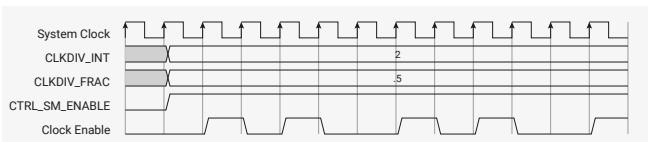
In general, an integer clock divisor of n will cause the state machine to run 1 cycle in every n , giving an effective clock speed of f_{sys} / n .

Figure 46. Integer clock divisors yield a periodic clock enable. The clock divider repeatedly counts down from n , and emits an enable pulse when it reaches 1.



Fractional division will maintain a steady state division rate of $n + f / 256$, where n and f are the integer and fractional fields of this state machine's `CLKDIV` register. It does this by selectively extending some division periods from n cycles to $n + 1$.

Figure 47. Fractional clock division with an average divisor of 2.5. The clock divider maintains a running total of the fractional value from each division period, and every time this value wraps through 1, the integer divisor is increased by one for the next division period.



For small n , the jitter introduced by a fractional divider may be unacceptable. However, for larger values, this effect is much less apparent.

NOTE

For fast asynchronous serial, it is recommended to use even divisions or multiples of 1 Mbaud where possible, rather than the traditional multiples of 300, to avoid unnecessary jitter.

3.5.6. GPIO Mapping

Internally, PIO has a 32-bit register for the output levels of each GPIO it can drive, and another register for the output enables (Hi/Lo-Z). On every system clock cycle, each state machine can write to some or all of the GPIOs in each of these registers.

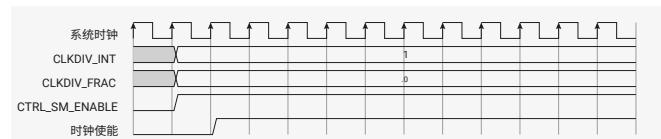
使状态机以某种稳定速率运行，速度可能远低于系统时钟。

通过这种方式实现时钟分频器，使状态机与系统之间的接口更简单、延迟更低且占用资源更小。当时钟使能为低时，状态机完全空闲，但系统仍可访问状态机的 FIFO 并更改其配置。

时钟分频器采用16位整数和8位小数表示，并使用一阶 delta-sigma 进行小数分频。时钟除数可在1至65536之间变化，增量为 $1 / 256$.

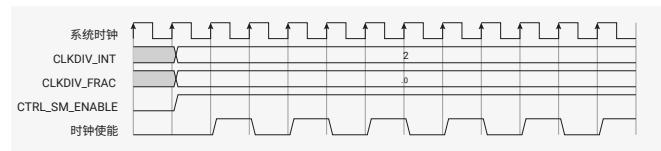
若时钟除数设置为1，则状态机会在每个周期运行，即全速运作：

图45。时钟除数为1时状态机的工作原理。一旦通过CTRL_SMR 寄存器启用状态机，其时钟使能将在每个周期被激活。



通常，整数时钟除数 n 使状态机每 n 周期运行一次，有效时钟频率为 f_{sys} / n .

图46。整数时钟除数产生周期性的时钟使能信号。时钟分频器不断从 n 递减计数，并在

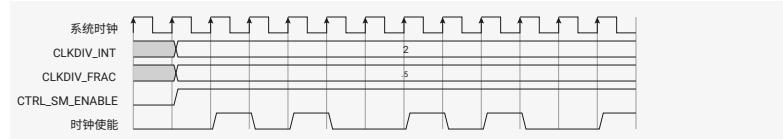


计数至 n 时发出使能脉冲。分数除法将保持稳定的分频速率为

$n + f / 256$ ，其中 n 与 f 分别表示该状态机 CLKDIV 寄存器中的整数与分数部分。其方式是通过选择性地将某些分频周期从一个周期延长至另一周期

$n + 1$.

图47。平均除数为2.5的分数时钟分频。时钟分频器维护每个分频周期分数值的累积总和，每当该值跨越1时，下一分频周期的整数除数即增加1。



对于小的 n ，分数分频器引入的抖动可能不可接受。然而，对于较大的数值，该效应则显著减弱。

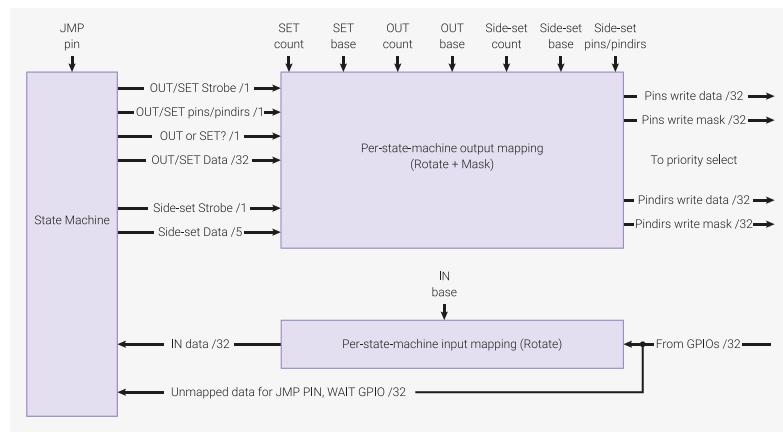
① 注意

对于高速异步串行，建议尽可能使用偶数分频或1 Mbaud的倍数，而非传统的300倍数，以避免不必要的抖动。

3.5.6. GPIO 映射

内部，PIO设有一个32位寄存器用于其可驱动的每个GPIO的输出电平，另有一个寄存器用于输出使能（高电平/高阻态）。在每个系统时钟周期内，每个状态机均可向这些寄存器中的部分或全部GPIO写入数据。

Figure 48. The state machine has two independent output channels, one shared by OUT/SET, and another used by side-set (which can happen at any time). Three independent mappings (first GPIO, number of GPIOs) control which GPIOs OUT, SET and side-set are directed to. Input data is rotated according to which GPIO is mapped to the LSB of the IN data.



The write data and write masks for the output level and output enable registers come from the following sources:

- An **OUT** instruction writes to up to 32 bits. Depending on the instruction's **Destination** field, this is applied to either pins or pindirs. The least-significant bit of **OUT** data is mapped to **PINCTRL_OUT_BASE**, and this mapping continues for **PINCTRL_OUT_COUNT** bits, wrapping after GPIO31.
- A **SET** instruction writes up to 5 bits. Depending on the instruction's **Destination** field, this is applied to either pins or pindirs. The least-significant bit of **SET** data is mapped to **PINCTRL_SET_BASE**, and this mapping continues for **PINCTRL_SET_COUNT** bits, wrapping after GPIO31.
- A side-set operation writes up to 5 bits. Depending on the register field **EXECCTRL_SIDE_PDIR**, this is applied to either pins or pindirs. The least-significant bit of side-set data is mapped to **PINCTRL_SIDESET_BASE**, continuing for **PINCTRL_SIDESET_COUNT** pins, minus one if **EXECCTRL_SIDE_EN** is set.

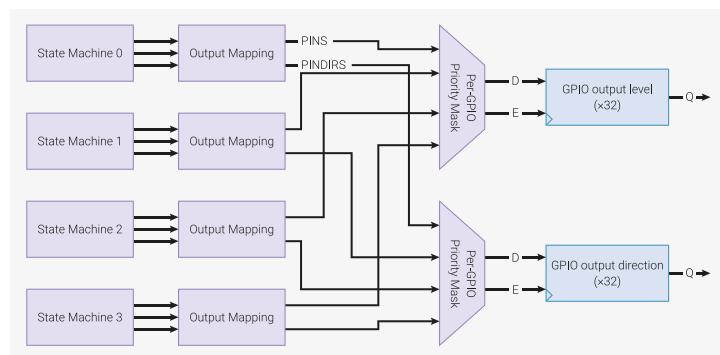
Each **OUT/SET**/side-set operation writes to a contiguous range of pins, but each of these ranges is independently sized and positioned in the 32-bit GPIO space. This is sufficiently flexible for many applications. For example, if one state machine is implementing some interface such as an SPI on a group of pins, another state machine can run the same program, mapped to a different group of pins, and provide a second SPI interface.

On any given clock cycle, the state machine may perform an **OUT** or a **SET**, and may simultaneously perform a side-set. The pin mapping logic generates a 32-bit write mask and write data bus for the output level and output enable registers, based on this request, and the pin mapping configuration.

If a side-set overlaps with an **OUT/SET** performed by that state machine on the same cycle, the side-set takes precedence in the overlapping region.

3.5.6.1. Output Priority

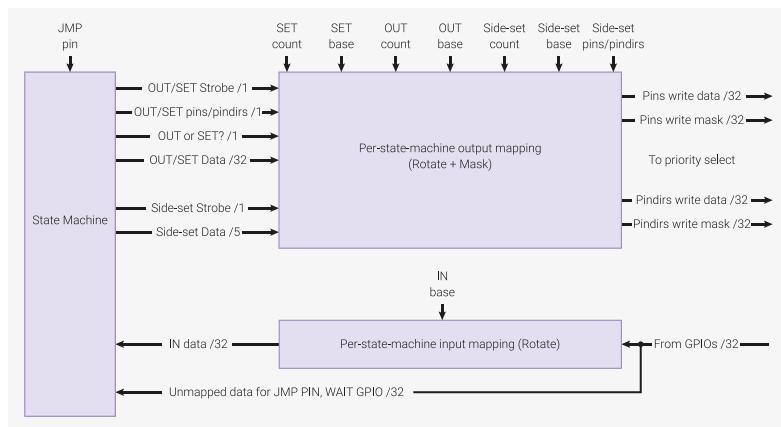
Figure 49. Per-GPIO priority select of write masks from each state machine. Each GPIO considers level and direction writes from each of the four state machines, and applies the value from the highest-numbered state machine.



Each state machine may assert an **OUT/SET** and a side-set through its pin mapping hardware on each cycle. This generates 32 bits of write data and write mask for the GPIO output level and output enable registers, from each state machine.

For each GPIO, PIO collates the writes from all four state machines, and applies the write from the highest-numbered

图48。状态机具有两个独立的输出通道，一个由OUT/SET共享，另一个用于side-set（可在任意时间发生）。三个独立映射（首个GPIO及GPIO数量）控制OUT、SET和side-set信号对应的GPIO端口。输入数据根据映射到IN数据最低有效位的GPIO进行循环移位。



用于输出电平和输出使能寄存器的写入数据及写入掩码来源如下：

- 一条 OUT 指令最多写入32位。依据指令的 Destination 字段，写入操作应用于引脚或引脚方向寄存器。OUT 数据的最低有效位映射至 PINCTRL_OUT_BASE，映射范围覆盖 PINCTRL_OUT_COUNT 位，超出 GPIO31 后循环映射。
- 一条 SET 指令最多写入5位。依据指令的 Destination 字段，写入操作应用于引脚或引脚方向寄存器。SET 数据的最低有效位映射至 PINCTRL_SET_BASE，映射范围覆盖 PINCTRL_SET_COUNT 位，超出 GPIO31 后循环映射。
- 侧置操作最多可写入5位。根据寄存器字段 EXECCTRL_SIDE_PINDIR，该操作应用于引脚或引脚方向。侧置数据的最低有效位映射至 PINCTRL_SIDESET_BASE，连续映射至 PINCTRL_SIDESET_COUNT 个引脚，若 EXECCTRL_SIDE_EN 被设置，则数量减一。

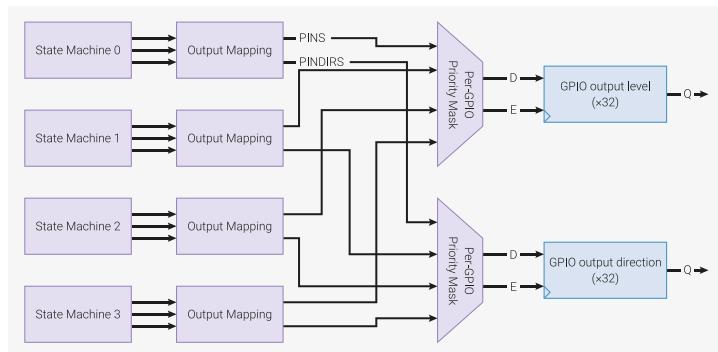
每个 OUT/SET/侧置操作写入一段连续的引脚范围，但这些范围在32位GPIO空间内的大小与位置均独立设定。这在多种应用场景中具备足够的灵活性。例如，若一个状态机在某组引脚上实现SPI等接口，另一个状态机可运行相同程序，映射至不同引脚组，从而提供第二个SPI接口。

在任一时钟周期内，状态机可能执行 OUT 或 SET 操作，同时亦可执行侧置操作。
引脚映射逻辑为输出电平及输出使能寄存器生成32位写入掩码及写入数据总线，
基于该请求及针脚映射配置。

如果旁路集（side-set）与同一状态机在同一周期执行的 OUT/SET 操作重叠，则旁路集在重叠区域内优先。

3.5.6.1. 输出优先级

图49。各状态机对每个GPIO写掩码的优先级选择。每个GPIO考虑四个状态机的电平和方向写入，并采用编号最高的状态机的值。



每个状态机可通过其针脚映射硬件在每个周期中断言一个 OUT/SET 信号和一个旁路集。这为每个状态机生成 GPIO 输出电平和输出使能寄存器的32位写数据及写掩码。

对于每个GPIO，PIO汇总来自所有四个状态机的写入信号，并采用编号最高的状态机的写入

state machine. This occurs separately for output levels and output directions – it is possible for a state machine to change both the level and direction of the same pin on the same cycle (e.g. via simultaneous `SET` and side-set), or for one state machine to change a GPIO's direction while another changes that GPIO's level. If no state machine asserts a write to a GPIO's level or direction, the value does not change.

There is a register stage between each state machine and the pin mapping logic, and a register stage between the input mapping logic and each state machine. Assuming zero propagation delay, a state machine observing its own outputs is subject to the following delays:

- when bypassing synchronisers, a two-cycle delay
- when synchronisers are engaged, a four-cycle delay

3.5.6.2. Input Mapping

The data observed by `IN` instructions is mapped such that the LSB is the GPIO selected by `PINCTRL_IN_BASE`, and successively more-significant bits come from successively higher-numbered GPIOs, wrapping after 31.

In other words, the `IN` bus is a right-rotate of the GPIO input values, by `PINCTRL_IN_BASE`. If fewer than 32 GPIOs are present, the PIO input is padded with zeroes up to 32 bits.

Some instructions, such as `WAIT GPIO`, use an absolute GPIO number, rather than an index into the `IN` data bus. In this case, the right-rotate is not applied.

3.5.6.3. Input Synchronisers

To protect PIO from metastabilities, each GPIO input is equipped with a standard 2-flipflop synchroniser. This adds two cycles of latency to input sampling, but the benefit is that state machines can perform an `IN PINS` at any point, and will see only a clean high or low level, not some intermediate value that could disturb the state machine circuitry. This is absolutely necessary for asynchronous interfaces such as UART RX.

It is possible to bypass these synchronisers, on a per-GPIO basis. This reduces input latency, but it is then up to the user to guarantee that the state machine does not sample its inputs at inappropriate times. Generally this is only possible for synchronous interfaces such as SPI. Synchronisers are bypassed by setting the corresponding bit in `INPUT_SYNC_BYPASS`.

WARNING

Sampling a metastable input can lead to unpredictable state machine behaviour. This should be avoided. Do not disable the synchronizers unless data applied to the pins meets setup and hold times relative to `CLK_SYS`.

3.5.7. Forced and EXEC'd Instructions

Besides the instruction memory, state machines can execute instructions from 3 other sources:

- `MOV EXEC` which executes an instruction from some register `Source`
- `OUT EXEC` which executes data shifted out from the OSR
- The `SMx_INSTR` control registers, to which the system can write instructions for immediate execution

```

1 .program exec_example
2
3 hang:
4     jmp hang
5 execute:
6     out exec, 32
7     jmp execute
8

```

内容。该过程分别针对输出电平和输出方向独立执行——同一周期内，状态机可同时更改同一引脚的电平和方向（例如通过同时的SET和旁路集），或由一个状态机更改GPIO方向，而另一个状态机更改该GPIO的电平。如果没有状态机声明写入GPIO的电平或方向，数值将保持不变。

每个状态机与引脚映射逻辑之间设有一级寄存器，输入映射逻辑与每个状态机之间亦设一级寄存器。假设传播延迟为零，状态机观察自身输出时存在如下延迟：

- 绕过同步器时，延迟为两个周期
- 启用同步器时，延迟为四个周期

3.5.6.2. 输入映射

IN指令观察到的数据映射为最低有效位对应由PINCTRL_IN_BASE选定的GPIO，且随后的更高有效位依次对应编号更高的GPIO，编号超过31后循环回绕。

换言之，IN总线即GPIO输入值按PINCTRL_IN_BASE所定义的值右旋转所得。若GPIO数量不足32个，PIO输入将以零补齐至32位。

某些指令，例如WAIT GPIO，使用绝对GPIO编号，而非对IN数据总线的索引。在这种情况下，右旋转操作不会被应用。

3.5.6.3. 输入同步器

为防止PIO出现亚稳态，每个GPIO输入均配备标准的两触发器同步器。这会增加两个周期的输入采样延迟，但其优点是状态机可在任意时刻执行IN PINS操作，并且仅会读取到干净的高或低电平，而非可能干扰状态机电路的中间状态。这对于诸如UART RX等异步接口而言是绝对必要的。

可以针对每个GPIO逐个选择绕过同步器。此举虽可减少输入延迟，但用户必须保证状态机不会在不适当的时间采样其输入信号。通常这仅适用于SPI等同步接口。通过设置INPUT_SYNC_BYPASS寄存器中的相应位即可绕过同步器。

● 警告

采样亚稳态输入可能导致状态机行为不可预测，应避免此类情况。除非应用于引脚的数据满足相对于CLK_SYS的建立时间和保持时间，否则不得禁用同步器。

3.5.7. 强制执行及 EXEC 指令

除了指令存储器外，状态机还可以从其他三个来源执行指令：

- MOV EXEC 执行来自寄存器 Source的指令
- OUT EXEC 执行从OSR移出的数据
- 系统可写入以供立即执行的SMx_INSTR控制寄存器

```

1 .program exec_example
2
3 hang:
4     jmp hang
5 execute:
6     out exec, 32
7     jmp execute
8

```

```

9 .program instructions_to_push
10
11     out x, 32
12     in x, 32
13     push

```

```

1 #include "tb.h" // TODO this is built against existing sw tree, so that we get printf etc
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "exec_example.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     for (int i = 0; i < count_of(exec_example_program); ++i)
15         mm_pio->instr_mem[i] = exec_example_program[i];
16
17     // Enable autopull, threshold of 32
18     mm_pio->sm[0].shiftctrl = (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB);
19
20     // Start state machine 0 -- will sit in "hang" loop
21     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
22
23     // Force a jump to program location 1
24     mm_pio->sm[0].instr = 0x0000 | 0x1; // jmp execute
25
26     // Feed a mixture of instructions and data into FIFO
27     mm_pio->txf[0] = instructions_to_push_program[0]; // out x, 32
28     mm_pio->txf[0] = 12345678;                         // data to be OUTed
29     mm_pio->txf[0] = instructions_to_push_program[1]; // in x, 32
30     mm_pio->txf[0] = instructions_to_push_program[2]; // push
31
32     // The program pushed into TX FIFO will return some data in RX FIFO
33     while (mm_pio->fstat & (1u << PIO_FSTAT_RXEMPTY_LSB))
34         ;
35
36     printf("%d\n", mm_pio->rx[0]);
37
38     return 0;
39 }

```

Here we load an example program into the state machine, which does two things:

- Enters an infinite loop
- Enters a loop which repeatedly pulls 32 bits of data from the TX FIFO, and executes the lower 16 bits as an instruction

The C program sets the state machine running, at which point it enters the `hang` loop. While the state machine is still running, the C program forces in a `jmp` instruction, which causes the state machine to break out of the loop.

When an instruction is written to the `INSTR` register, the state machine immediately decodes and executes that instruction, rather than the instruction it would have fetched from the PIO's instruction memory. The program counter does not advance, so on the next cycle (assuming the instruction forced into the `INSTR` interface did not stall) the state machine continues to execute its current program from the point where it left off, unless the written instruction itself

```

9 .program instructions_to_push
10
11     out x, 32
12     in x, 32
13     push

```

```

1 #include "tb.h" // TODO 本文件基于现有软件架构构建, 以支持 printf 等功能
2
3 #include "platform.h"
4 #include "pio_regs.h"
5 #include "system.h"
6 #include "hardware.h"
7
8 #include "exec_example.pio.h"
9
10 int main()
11 {
12     tb_init();
13
14     for (int i = 0; i < count_of(exec_example_program); ++i)
15         mm_pio->instr_mem[i] = exec_example_program[i];
16
17     // 启用自动拉取, 阈值设置为 32
18     mm_pio->sm[0].shiftctrl = (1u << PIO_SM0_SHIFTCTRL_AUTOPULL_LSB);
19
20     // 启动状态机0 —— 将停留在 “hang” 循环
21     hw_set_bits(&mm_pio->ctrl, 1u << (PIO_CTRL_SM_ENABLE_LSB + 0));
22
23     // 强制跳转至程序位置 1
24     mm_pio->sm[0].instr = 0x0000 | 0x1; // 执行 jmp 指令
25
26     // 向 FIFO 输入混合指令和数据
27     mm_pio->txf[0] = instructions_to_push_program[0]; // 输出 x, 32 位
28     mm_pio->txf[0] = 12345678; // 待 OUT 的数据
29     mm_pio->txf[0] = instructions_to_push_program[1]; // 在 x 寄存器中, 32
30     mm_pio->txf[0] = instructions_to_push_program[2]; // push
31
32     // 推入 TX FIFO 的程序将在 RX FIFO 返回部分数据
33     while (mm_pio->fstat & (1u << PIO_FSTAT_RXEMPTY_LSB))
34     ;
35
36     printf("%d\n", mm_pio->rx[0]);
37
38     return 0;
39 }

```

此处我们将示例程序加载入状态机，该程序执行以下两项操作：

- 进入无限循环
- 进入一个循环，反复从 TX FIFO 拉取 32 位数据，并将低 16 位作为指令执行

C 程序启动状态机，此时状态机进入 `hang` 循环状态。在状态机仍运行时，C 程序强制插入一个 `jmp` 指令，导致状态机跳出循环。

当指令写入 `INSTR` 寄存器时，状态机会立即解码并执行该指令，而非执行原本应从 PIO 指令存储器中取出的指令。程序计数器不会前进，因此在下一周期（假设写入 `INSTR` 接口的指令未阻塞），状态机会从暂停处继续执行当前程序，除非写入的指令本身

manipulated **PC**.

Delay cycles are ignored on instructions written to the **INSTR** register, and execute immediately, ignoring the state machine clock divider. This interface is provided for performing initial setup and effecting control flow changes, so it executes instructions in a timely manner, no matter how the state machine is configured.

Instructions written to the **INSTR** register are permitted to stall, in which case the state machine will latch this instruction internally until it completes. This is signified by the **EXECCTRL_EXEC_STALLED** flag. This can be cleared by restarting the state machine, or writing a **NOP** to **INSTR**.

In the second phase of the example state machine program, the **OUT EXEC** instruction is used. The **OUT** itself occupies one execution cycle, and the instruction which the **OUT** executes is on the next execution cycle. Note that one of the instructions we execute is also an **OUT** — the state machine is only capable of executing one **OUT** instruction on any given cycle.

OUT EXEC works by writing the **OUT** shift data to an internal instruction latch. On the next cycle, the state machine remembers it must execute from this latch rather than the instruction memory, and also knows to not advance **PC** on this second cycle.

This program will print "12345678" when run.

⚠ CAUTION

If an instruction written to **INSTR** stalls, it is stored in the same instruction latch used by **OUT EXEC** and **MOV EXEC**, and will overwrite an in-progress instruction there. If **EXEC** instructions are used, instructions written to **INSTR** must not stall.

3.6. Examples

These examples illustrate some of PIO's hardware features, by implementing common I/O interfaces.

Looking to get started?

The [Raspberry Pi Pico-series C/C++ SDK](#) book has a comprehensive PIO chapter, which walks through writing and building a first PIO application, and goes on to walk through some programs line-by-line. It also covers broader topics such as using PIO with DMA, and goes into much more depth on how PIO can be integrated into your software.

3.6.1. Duplex SPI

操控了 **PC**。

写入 **INSTR**寄存器的指令将忽略延迟周期，立即执行，并跳过状态机时钟分频器。此接口用于执行初始配置及控制流变更，因此无论状态机配置如何，都能及时执行指令。

写入 **INSTR**寄存器的指令允许发生阻塞，状态机会在内部锁存该指令直至完成。此状态由**EXECCTRL_EXEC_STALLED**标志指示。该标志可通过重启状态机或向**INSTR**写入 **NOP**指令予以清除。

在示例状态机程序的第二阶段，使用**OUT EXEC**指令。**OUT**本身占用一个执行周期，而 **OUT**所执行的指令位于下一个执行周期。请注意，我们执行的指令之一也是一个 **OUT** —— 状态机在任何周期内仅能执行一个 **OUT**指令。

OUT EXEC通过将 **OUT**移位数据写入内部指令锁存器来实现。在下一个周期，状态机会记住必须从该锁存器而非指令存储器执行，并且知晓该第二周期内不应递增 **PC**。

该程序运行时将打印"12345678"。

⚠ 注意

若写入 **INSTR**的指令阻塞，则其被存储于由**OUT EXEC**和**MOV EXEC**共用的指令锁存器中，且会覆盖锁存器内正在执行的指令。若使用 **EXEC**指令，写入 **INSTR**的指令不得阻塞。

3.6. 示例

这些示例通过实现常见的输入输出接口，展示了PIO的部分硬件特性。

准备开始吗？

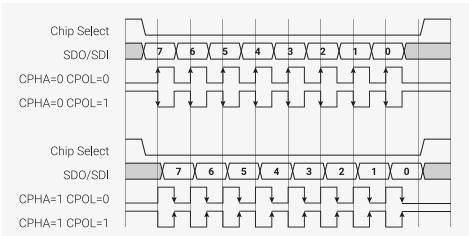
《Raspberry Pi Pico系列C/C++ SDK》一书包含了详尽的PIO章节，逐步讲解如何编写和构建首个PIO应用程序，并对部分程序逐行解析。该章节还涉及使用PIO与DMA的更广泛主题，并深入探讨了PIO

如何集成到您的软件中。

3.6.1. 双工 SPI

Figure 50. In SPI, a host and device exchange data over a bidirectional pair of serial data lines, synchronous with a clock (SCK). Two flags, CPOL and CPHA, specify the clock's behaviour.

CPOL is the idle state of the clock: 0 for low, 1 for high. The clock pulses a number of times, transferring one bit in each direction per pulse, but always returns to its idle state. CPHA determines on which edge of the clock data is captured: 0 for leading edge, and 1 for trailing edge. The arrows in the figure show the clock edge where data is captured by both the host and device.



SPI is a common serial interface with a twisty history. The following program implements full-duplex (i.e. transferring data in both directions simultaneously) SPI, with a CPHA parameter of 0.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> Lines 14 - 32

```

14 .program spi_cpha0
15 .side_set 1
16
17 ; Pin assignments:
18 ; - SCK is side-set pin 0
19 ; - MOSI is OUT pin 0
20 ; - MISO is IN pin 0
21 ;
22 ; Autopush and autopull must be enabled, and the serial frame size is set by
23 ; configuring the push/pull threshold. Shift left/right is fine, but you must
24 ; justify the data yourself. This is done most conveniently for frame sizes of
25 ; 8 or 16 bits by using the narrow store replication and narrow load byte
26 ; picking behaviour of RP2040's IO fabric.
27
28 ; Clock phase = 0: data is captured on the leading edge of each SCK pulse, and
29 ; transitions on the trailing edge, or some time before the first leading edge.
30
31     out pins, 1 side 0 [1] ; Stall here on empty (sideset proceeds even if
32     in pins, 1 side 1 [1] ; instruction stalls, so we stall with SCK low)

```

This code uses autopush and autopull to continuously stream data from the FIFOs. The entire program runs once for every bit that is transferred, and then loops. The state machine tracks how many bits have been shifted in/out, and automatically pushes/pulls the FIFOs at the correct point. A similar program handles the CPHA=1 case:

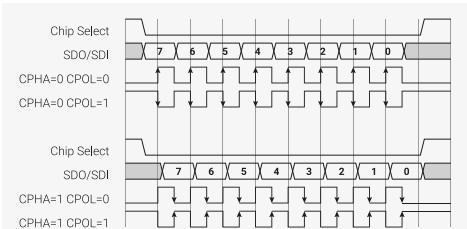
Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> Lines 34 - 42

```

34 .program spi_cpha1
35 .side_set 1
36
37 ; Clock phase = 1: data transitions on the leading edge of each SCK pulse, and
38 ; is captured on the trailing edge.
39
40     out x, 1    side 0      ; Stall here on empty (keep SCK deasserted)
41     mov pins, x side 1 [1] ; Output data, assert SCK (mov pins uses OUT mapping)
42     in pins, 1 side 0      ; Input data, deassert SCK

```

图50。在SPI通信中，主机和设备通过一对双向串行数据线交换数据，数据传输与时钟（SCK）同步进行。两个标志位CPOL和CPHA用于定义时钟的工作。CPOL表示时钟的空闲状态：0为低电平，1为高电平。时钟会发出多个脉冲，每个脉冲在两个方向传输一位数据，但始终回归空闲状态。CPHA决定在时钟的哪个边沿采样数据：0表示前沿，1表示后沿。图中箭头表示主机和设备均在该时钟边沿捕获数据。



SPI是一种常见的串行接口，其发展历史较为复杂。以下程序实现了全双工（即同时双向传输数据）的SPI，CPHA参数设为0。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第14至32行

```

14 .program spi_cpha0
15 .side_set 1
16
17 ; 引脚分配：
18 ; - SCK 为 side-set 引脚 0
19 ; - MOSI 为 OUT 引脚 0
20 ; - MISO 为 IN 引脚 0
21 ;
22 ; 必须启用自动推送和自动拉取，串行帧大小通过
23 ; 配置推送/拉取阈值来设定。左/右移位操作可用，但需
24 ; 自行调整数据对齐。这对于帧大小为
25 ; 通过使用窄存储复制和窄加载字节实现25; 8位或16位
26 ; RP2040的IO结构的选取行为。
27
28 ; 时钟相位 = 0: 数据在每个SCK脉冲的上升沿采集，且
29 ; 在下降沿发生转换，或在第一个上升沿之前某个时间点发生转换。
30
31     输出引脚, 1侧0 [1]; 此处空时停顿（即使侧设继续执行,
32     输入引脚, 1侧1 [1]; 指令停顿，因此我们在SCK低电平时停顿)

```

该代码使用autopush和autopull持续从FIFO流式传输数据。整个程序针对传输的每个位执行一次，然后循环执行。状态机跟踪已移入/移出的位数，并在适当时机自动推入/拉出FIFO。类似程序用于处理CPHA=1的情况：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第34至42行

```

34 .program spi_cpha1
35 .side_set 1
36
37 ; 时钟相位 = 1: 数据在每个 SCK 脉冲的上升沿发生变化,
38 ; 并在下降沿被捕获。
39
40     out x, 1      side 0      ; 空时暂停（保持 SCK 处于非激活状态）
41     mov pins, x side 1 [1] ; 输出数据，同时置位 SCK (mov pins 使用 OUT 映射)
42     in pins, 1    side 0      ; 输入数据，复位 SCK

```

NOTE

These programs do not control the chip select line; chip select is often implemented as a software-controlled GPIO, due to wildly different behaviour between different SPI hardware. The full `spi.pio` source linked above contains some examples how PIO can implement a hardware chip select line.

A C helper function configures the state machine, connects the GPIOs, and sets the state machine running. Note that the SPI frame size – that is, the number of bits transferred for each FIFO record – can be programmed to any value from 1 to 32, without modifying the program. Once configured, the state machine is set running.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> Lines 46 - 72

```

46 static inline void pio_spi_init(PIO pio, uint sm, uint prog_offs, uint n_bits,
47     float clkdiv, bool cpha, bool cpol, uint pin_sck, uint pin_mosi, uint pin_miso) {
48     pio_sm_config c = cpha ? spi_cpha1_program_get_default_config(prog_offs) :
49         spi_cpha0_program_get_default_config(prog_offs);
50     sm_config_set_out_pins(&c, pin_mosi, 1);
51     sm_config_set_in_pins(&c, pin_miso);
52     sm_config_set_sideset_pins(&c, pin_sck);
53     // Only support MSB-first in this example code (shift to left, auto push/pull,
54     threshold=nbits)
55     sm_config_set_out_shift(&c, false, true, n_bits);
56     sm_config_set_in_shift(&c, false, true, n_bits);
57     sm_config_set_clkdiv(&c, clkdiv);
58
59     // MOSI, SCK output are low, MISO is input
60     pio_sm_set_pins_with_mask(pio, sm, 0, (1u << pin_sck) | (1u << pin_mosi));
61     pio_sm_set_pindirs_with_mask(pio, sm, (1u << pin_sck) | (1u << pin_mosi), (1u << pin_sck)
62         | (1u << pin_mosi) | (1u << pin_miso));
63     pio_gpio_init(pio, pin_mosi);
64     pio_gpio_init(pio, pin_miso);
65     pio_gpio_init(pio, pin_sck);
66
67     // The pin muxes can be configured to invert the output (among other things
68     // and this is a cheesy way to get CPOL=1
69     gpio_set_outr(pin_sck, cpol ? GPIO_OVERRIDE_INVERT : GPIO_OVERRIDE_NORMAL);
70     // SPI is synchronous, so bypass input synchroniser to reduce input delay.
71     hw_set_bits(&pio->input_sync_bypass, 1u << pin_miso);
72
73     pio_sm_init(pio, sm, prog_offs, &c);
74     pio_sm_set_enabled(pio, sm, true);
75 }
```

The state machine will now immediately begin to shift out any data appearing in the TX FIFO, and push received data into the RX FIFO.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/pio_spi.c Lines 18 - 34

```

18 void __time_critical_func(pio_spi_write8_blocking)(const pio_spi_inst_t *spi, const uint8_t
    *src, size_t len) {
19     size_t tx_remain = len, rx_remain = len;
20     // Do 8 bit accesses on FIFO, so that write data is byte-replicated. This
21     // gets us the left-justification for free (for MSB-first shift-out)
22     io_rw_8 *txfifo = (io_rw_8 *) &spi->pio->txf[spi->sm];
23     io_rw_8 *rxfifo = (io_rw_8 *) &spi->pio->rx[spi->sm];
24     while (tx_remain || rx_remain) {
25         if (tx_remain && !pio_sm_is_tx_fifo_full(spi->pio, spi->sm)) {
26             *txfifo = *src++;
27             --tx_remain;
28         }
29 }
```

注意

这些程序不控制片选线；由于不同 SPI 硬件行为差异显著，片选通常由软件控制的 GPIO 实现。上述链接中的完整 `spi.pio` 源代码包含了 PIO 实现硬件片选线的示例。

C 语言辅助函数用于配置状态机、连接 GPIO 并启动状态机运行。请注意，SPI 帧大小——即每个 FIFO 记录传输的位数——可编程为 1 至 32 之间的任意值，无需修改程序。配置完成后，状态机即开始运行。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi.pio> 第46至72行

```

46 static inline void pio_spi_init(PIO pio, uint sm, uint prog_offs, uint n_bits,
47     float clkdiv, bool cpha, bool cpol, uint pin_sck, uint pin_mosi, uint pin_miso) {
48     pio_sm_config c = cpha ? spi_cpha1_program_get_default_config(prog_offs) :
49         spi_cpha0_program_get_default_config(prog_offs);
50     sm_config_set_out_pins(&c, pin_mosi, 1);
51     sm_config_set_in_pins(&c, pin_miso);
52     sm_config_set_sideset_pins(&c, pin_sck);
53     // 本示例代码仅支持最高有效位优先（左移，自动推送/拉取，阈值=nbits）

54     sm_config_set_out_shift(&c, false, true, n_bits);
55     sm_config_set_in_shift(&c, false, true, n_bits);
56     sm_config_set_clkdiv(&c, clkdiv);

57     // MOSI、SCK输出为低电平，MISO为输入
58     pio_sm_set_pins_with_mask(pio, sm, 0, (1u << pin_sck) | (1u << pin_mosi));
59     pio_sm_set_pindirs_with_mask(pio, sm, (1u << pin_sck) | (1u << pin_mosi), (1u << pin_sck)
60     | (1u << pin_mosi) | (1u << pin_miso));
61     pio_gpio_init(pio, pin_mosi);
62     pio_gpio_init(pio, pin_miso);
63     pio_gpio_init(pio, pin_sck);

64     // 引脚多路复用器可被配置为反转输出（以及其他功能）
65     // 这是一种使CPOL=1 的简便方法
66     gpio_set_outover(pin_sck, cpol ? GPIO_OVERRIDE_INVERT : GPIO_OVERRIDE_NORMAL);
67     // SPI 是同步的，因此绕过输入同步器以降低输入延迟。
68     hw_set_bits(&pio->input_sync_bypass, 1u << pin_miso);
69
70     pio_sm_init(pio, sm, prog_offs, &c);
71     pio_sm_set_enabled(pio, sm, true);
72 }
```

状态机将立即开始输出 TX FIFO 中的任何数据，并将接收的数据推入 RX FIFO。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/pio_spi.c 第18至34行

```

18 void __time_critical_func(pio_spi_write8_blocking)(const pio_spi_inst_t *spi, const uint8_t
    *src, size_t len) {
19     size_t tx_remain = len, rx_remain = len;
20     // 对 FIFO 进行 8 位访问，使写入数据按字节复制。
21     // 该操作可免费实现左对齐（针对最高有效位优先移位输出）
22     io_rw_8 *txfifo = (io_rw_8 *) &spi->pio->txf[spi->sm];
23     io_rw_8 *rxfifo = (io_rw_8 *) &spi->pio->rxf[spi->sm];
24     while (tx_remain || rx_remain) {
25         if (tx_remain && !pio_sm_is_tx_fifo_full(spi->pio, spi->sm)) {
26             *txfifo = *src++;
27             --tx_remain;
28         }
29     }
30 }
```

```

29     if (rx_remain && !pio_sm_is_rx_fifo_empty(spi->pio, spi->sm)) {
30         (void) *rxfifo;
31         --rx_remain;
32     }
33 }
34 }
```

Putting this all together, this complete C program will loop back some data through a PIO SPI at 1MHz, with all four CPOL/CPHA combinations:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi_loopback.c

```

1 /**
2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "pio_spi.h"
12
13 // This program instantiates a PIO SPI with each of the four possible
14 // CPOL/CPHA combinations, with the serial input and output pin mapped to the
15 // same GPIO. Any data written into the state machine's TX FIFO should then be
16 // serialised, deserialised, and reappear in the state machine's RX FIFO.
17
18 #define PIN_SCK 18
19 #define PIN_MOSI 16
20 #define PIN_MISO 16 // same as MOSI, so we get loopback
21
22 #define BUF_SIZE 20
23
24 void test(const pio_spi_inst_t *spi) {
25     static uint8_t txbuf[BUF_SIZE];
26     static uint8_t rxbuf[BUF_SIZE];
27     printf("TX:");
28     for (int i = 0; i < BUF_SIZE; ++i) {
29         txbuf[i] = rand() >> 16;
30         rxbuf[i] = 0;
31         printf(" %02x", (int) txbuf[i]);
32     }
33     printf("\n");
34
35     pio_spi_write8_read8_blocking(spi, txbuf, rxbuf, BUF_SIZE);
36
37     printf("RX:");
38     bool mismatch = false;
39     for (int i = 0; i < BUF_SIZE; ++i) {
40         printf(" %02x", (int) rxbuf[i]);
41         mismatch = mismatch || rxbuf[i] != txbuf[i];
42     }
43     if (mismatch)
44         printf("\n Nope\n");
45     else
46         printf("\n OK\n");
47 }
48
49 int main() {
50     stdio_init_all();
```

```

29         if (rx_remain && !pio_sm_is_rx_fifo_empty(spi->pio, spi->sm)) {
30             (void) *rxfifo;
31             --rx_remain;
32         }
33     }
34 }
```

综合以上内容，该完整的C程序将通过一个1MHz的PIO SPI进行数据回环，覆盖所有四种CPOL/CPHA组合：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/spi/spi_loopback.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4  * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "pio_spi.h"
12
13 // 本程序实例化了每种可能的
14 // CPOL/CPHA组合的PIO SPI，串行输入和输出引脚映射为
15 // 相同的GPIO。写入状态机TX FIFO的任何数据
16 // 应被串行化、反序列化，并重新出现在状态机的RX FIFO中。
17
18 #define PIN_SCK 18
19 #define PIN_MOSI 16
20 #define PIN_MISO 16 // 与MOSI相同，实现回环
21
22 #define BUF_SIZE 20
23
24 void test(const pio_spi_inst_t *spi) {
25     static uint8_t txbuf[BUF_SIZE];
26     static uint8_t rdbuf[BUF_SIZE];
27     printf("TX:");
28     for (int i = 0; i < BUF_SIZE; ++i) {
29         txbuf[i] = rand() >> 16;
30         rdbuf[i] = 0;
31         printf(" %02x", (int)txbuf[i]);
32     }
33     printf("\n");
34
35     pio_spi_write8_read8_blocking(spi, txbuf, rdbuf, BUF_SIZE);
36
37     printf("RX:");
38     bool mismatch = false;
39     for (int i = 0; i < BUF_SIZE; ++i) {
40         printf(" %02x", (int)rdbuf[i]);
41         mismatch = mismatch || rdbuf[i] != txbuf[i];
42     }
43     if (mismatch)
44         printf("\n不匹配\n");
45     else
46         printf("\n成功\n");
47 }
48
49 int main() {
50     stdio_init_all();
```

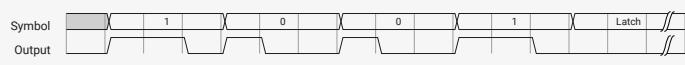
```

51
52     pio_spi_inst_t spi = {
53         .pio = pio0,
54         .sm = 0
55     };
56     float clkdiv = 31.25f; // 1 MHz @ 125 clk_sys
57     uint cpha0_prog_offs = pio_add_program(spi.pio, &spi_cpha0_program);
58     uint cpha1_prog_offs = pio_add_program(spi.pio, &spi_cpha1_program);
59
60     for (int cpha = 0; cpha <= 1; ++cpha) {
61         for (int cpol = 0; cpol <= 1; ++cpol) {
62             printf("CPHA = %d, CPOL = %d\n", cpha, cpol);
63             pio_spi_init(spi.pio, spi.sm,
64                         cpha ? cpha1_prog_offs : cpha0_prog_offs,
65                         8,           // 8 bits per SPI frame
66                         clkdiv,
67                         cpha,
68                         cpol,
69                         PIN_SCK,
70                         PIN_MOSI,
71                         PIN_MISO
72                     );
73             test(&spi);
74             sleep_ms(10);
75         }
76     }
77 }
```

3.6.2. WS2812 LEDs

WS2812 LEDs are driven by a proprietary pulse-width serial format, with a wide positive pulse representing a "1" bit, and narrow positive pulse a "0". Each LED has a serial input and a serial output; LEDs are connected in a chain, with each serial input connected to the previous LED's serial output.

Figure 51. WS2812 line format. Wide positive pulse for 1, narrow positive pulse for 0, very long negative pulse for latch enable



LEDs consume 24 bits of pixel data, then pass any additional input data on to their output. In this way a single serial burst can individually program the colour of each LED in a chain. A long negative pulse latches the pixel data into the LEDs.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> Lines 8 - 31

```

8 .program ws2812
9 .side_set 1
10
11 ; The following constants are selected for broad compatibility with WS2812,
12 ; WS2812B, and SK6812 LEDs. Other constants may support higher bandwidths for
13 ; specific LEDs, such as (7,10,8) for WS2812B LEDs.
14
15 .define public T1 3
16 .define public T2 3
17 .define public T3 4
18
19 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
20 .lang_opt python out_init      = pico.PIO.OUT_HIGH
21 .lang_opt python out_shiftdir = 1
22
23 .wrap_target
```

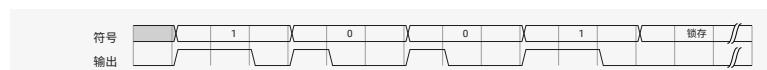
```

51
52     pio_spi_inst_t spi = {
53         .pio = pio0,
54         .sm = 0
55     };
56     float clkdiv = 31.25f; // 1 MHz @ 125 clk_sys
57     uint cpha0_prog_offs = pio_add_program(spi.pio, &spi_cpha0_program);
58     uint cpha1_prog_offs = pio_add_program(spi.pio, &spi_cpha1_program);
59
60     for (int cpha = 0; cpha <= 1; ++cpha) {
61         for (int cpol = 0; cpol <= 1; ++cpol) {
62             printf("CPHA = %d, CPOL = %d\n", cpha, cpol);
63             pio_spi_init(spi.pio, spi.sm,
64                         cpha ? cpha1_prog_offs : cpha0_prog_offs,
65                         8,           // 每个 SPI 帧 8 位
66                         clkdiv,
67                         cpha,
68                         cpol,
69                         PIN_SCK,
70                         PIN_MOSI,
71                         PIN_MISO
72                     );
73             test(&spi);
74             sleep_ms(10);
75         }
76     }
77 }
```

3.6.2. WS2812 LED

WS2812 LED 由专有脉宽串行格式驱动，宽正脉冲表示“1”位，窄正脉冲表示“0”。每个 LED 拥有一个串行输入和一个串行输出；LED 以链式连接，每个串行输入连接到前一 LED 的串行输出。

图 51. WS2812 线路格式。宽正脉冲表示1，窄正脉冲表示0，极长负脉冲表示锁存使能



LED 消耗 24 位像素数据，随后将任何额外输入数据传递到其输出。通过此方式，单个串行数据流可单独编程链中每个 LED 的颜色。一个较长的负脉冲将像素数据锁存至 LED。

Pico 示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> 第8行至第31行

```

8 .program ws2812
9 .side_set 1
10
11 ; 以下常量被选以确保与WS2812、
12 ; WS2812B及SK6812 LED的广泛兼容。其他常量可能支持
13 ; 特定LED的更高带宽，如WS2812B LED所用的(7, 10, 8)常量。
14
15 .define public T1 3
16 .define public T2 3
17 .define public T3 4
18
19 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
20 .lang_opt python out_init      = pico.PIO.OUT_HIGH
21 .lang_opt python out_shiftdir = 1
22
23 .wrap_target
```

```

24 bitloop:
25     out x, 1      side 0 [T3 - 1] ; Side-set still takes place when instruction stalls
26     jmp !x do_zero side 1 [T1 - 1] ; Branch on the bit we shifted out. Positive pulse
27 do_one:
28     jmp bitloop    side 1 [T2 - 1] ; Continue driving high, for a long pulse
29 do_zero:
30     nop           side 0 [T2 - 1] ; Or drive low, for a short pulse
31 .wrap

```

This program shifts bits from the OSR into X, and produces a wide or narrow pulse on side-set pin 0, based on the value of each data bit. Autopull must be configured, with a threshold of 24. Software can then write 24-bit pixel values into the FIFO, and these will be serialised to a chain of WS2812 LEDs. The `.pio` file contains a C helper function to set this up:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> Lines 36 - 52

```

36 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
37     bool rgbw) {
38     pio_gpio_init(pio, pin);
39     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
40
41     pio_sm_config c = ws2812_program_get_default_config(offset);
42     sm_config_set_sideset_pins(&c, pin);
43     sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
44     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
45
46     int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
47     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
48     sm_config_set_clkdiv(&c, div);
49
50     pio_sm_init(pio, sm, offset, &c);
51     pio_sm_set_enabled(pio, sm, true);
52 }

```

Because the shift is MSB-first, and our pixels aren't a power of two size (so we can't rely on the narrow write replication behaviour on RP2040 to fan out the bits for us), we need to preshift the values written to the TX FIFO.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c> Lines 43 - 45

```

43 static inline void put_pixel(PIO pio, uint sm, uint32_t pixel_grb) {
44     pio_sm_put_blocking(pio, sm, pixel_grb << 8u);
45 }

```

To DMA the pixels, we could instead set the autopull threshold to 8 bits, set the DMA transfer size to 8 bits, and write a byte at a time into the FIFO. Each pixel would be 3 one-byte transfers. Because of how the bus fabric and DMA on RP2040 work, each byte the DMA transfers will appear replicated four times when written to a 32-bit IO register, so effectively your data is at *both ends* of the shift register, and you can shift in either direction without worry.

More detail?

The WS2812 example is the subject of a tutorial in the [Raspberry Pi Pico-series C/C++ SDK](#) document, in the PIO chapter. The tutorial dissects the `ws2812` program line by line, traces through how the program executes, and shows wave diagrams of the GPIO output at every point in the program.

```

24 bitloop:
25     out x, 1           side 0 [T3 - 1] ;当指令停顿时，侧边设定仍会发生
26     jmp !x do_zero side 1 [T1 - 1] ;根据移出位进行跳转。正脉冲
27 do_one:
28     jmp bitloop    side 1 [T2 - 1] ;继续保持高电平，形成长脉冲
29 do_zero:
30     nop            side 0 [T2 - 1] ;或保持低电平，形成短脉冲
31 .wrap

```

该程序将OSR中的位移入X寄存器，并根据每个位数据值，在侧边设定引脚0上产生宽脉冲或窄脉冲。必须配置自动拉取（autopull），阈值设为24。软件随后可向FIFO写入24位像素值，数据将被串行发送至一串WS2812 LED。该.pio文件包含用于配置此功能的C语言辅助函数：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.pio> 第36行至52行

```

36 static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq,
37                                         bool rgbw) {
38     pio_gpio_init(pio, pin);
39     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
40
41     pio_sm_config c = ws2812_program_get_default_config(offset);
42     sm_config_set_sideset_pins(&c, pin);
43     sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
44     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
45
46     int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
47     float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
48     sm_config_set_clkdiv(&c, div);
49
50     pio_sm_init(pio, sm, offset, &c);
51     pio_sm_set_enabled(pio, sm, true);
52 }

```

由于移位按最高有效位优先进行，且我们的像素尺寸非二的幂次（因此无法依赖RP2040上的窄写复制行为来展开位），故需对写入TX FIFO的值进行预移位。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c> 第43至45行

```

43 static inline void put_pixel(PIO pio, uint sm, uint32_t pixel_grb) {
44     pio_sm_put_blocking(pio, sm, pixel_grb << 8u);
45 }

```

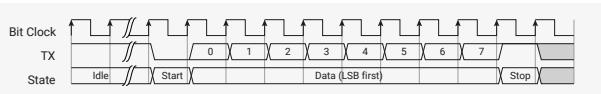
若采用DMA传输像素，则可将autopull阈值设为8位，DMA传输单位设为8位，并每次向FIFO写入一个字节。每个像素将由3次单字节传输组成。由于RP2040上的总线结构和DMA的工作原理，DMA传输的每个字节在写入32位IO寄存器时会被复制四次，因此数据实际上位于移位寄存器的两端，您可以放心地向任意方向进行移位。

需要更详细的信息吗？

WS2812示例是Raspberry Pi Pico系列C/C++ SDK文档中PIO章节的教程内容。该教程逐行解析ws2812程序，追踪程序的执行过程，并展示程序各阶段GPIO输出的波形图。

3.6.3. UART TX

Figure 52. UART serial format. The line is high when idle. The transmitter pulls the line down for one bit period to signify the start of a serial frame (the "start bit"), and a small, fixed number of data bits follows. The line returns to the idle state for at least one bit period (the "stop bit") before the next serial frame can begin.



This program implements the transmit component of a universal asynchronous receive/transmit (UART) serial peripheral. Perhaps it would be more correct to refer to this as a UAT.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio Lines 8 - 18

```

8 .program uart_tx
9 .side_set 1 opt
10
11 ; An 8n1 UART transmit program.
12 ; OUT pin 0 and side-set pin 0 are both mapped to UART TX pin.
13
14     pull      side 1 [7] ; Assert stop bit, or stall with line in idle state
15     set x, 7    side 0 [7] ; Preload bit counter, assert start bit for 8 clocks
16     bitloop:           ; This loop will run 8 times (8n1 UART)
17     out pins, 1        ; Shift 1 bit from OSR to the first OUT pin
18     jmp x-- bitloop [6] ; Each loop iteration is 8 cycles.

```

As written, it will:

- Stall with the pin driven high until data appears (noting that side-set takes effect even when the state machine is stalled)
- Assert a start bit, for 8 SM execution cycles
- Shift out 8 data bits, each lasting for 8 cycles
- Return to the idle line state for at least 8 cycles before asserting the next start bit

If the state machine's clock divider is configured to run at 8 times the desired baud rate, this program will transmit well-formed UART serial frames, whenever data is pushed to the TX FIFO either by software or the system DMA. To extend the program to cover different frame sizes (different numbers of data bits), the `set x, 7` could be replaced with `mov x, y`, so that the `y` scratch register becomes a per-SM configuration register for UART frame size.

The `.pio` file in the SDK also contains this function, for configuring the pins and the state machine, once the program has been loaded into the PIO instruction memory:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio Lines 24 - 51

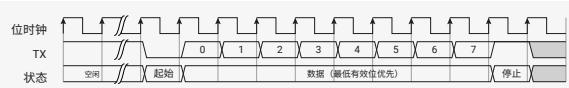
```

24 static inline void uart_tx_program_init(PIO pio, uint sm, uint offset, uint pin_tx, uint
25   baud) {
26   // Tell PIO to initially drive output-high on the selected pin, then map PIO
27   // onto that pin with the IO muxes.
28   pio_sm_set_pins_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
29   pio_sm_set_pindirs_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
30   pio_gpio_init(pio, pin_tx);
31
32   pio_sm_config c = uart_tx_program_get_default_config(offset);
33
34   // OUT shifts to right, no autopull
35   sm_config_set_out_shift(&c, true, false, 32);
36
37   // We are mapping both OUT and side-set to the same pin, because sometimes
38   // we need to assert user data onto the pin (with OUT) and sometimes
39   // assert constant values (start/stop bit)
40   sm_config_set_out_pins(&c, pin_tx, 1);
41   sm_config_set_sideset_pins(&c, pin_tx);

```

3.6.3. UART 发送

图52. UART串行格式。线路空闲时电平为高。发送端将线路拉低一个比特周期以标志串行帧开始（“起始位”），随后传输固定数量的数据位。线路在下一个串行帧开始之前至少保持一个比特周期（“停止位”）的空闲状态。



本程序实现了通用异步收发器（UART）串行外设的发送部分。或许可更准确地称其为UAT。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio，第8至18行

```
8 .program uart_tx
9 .side_set 1 opt
10
11 ; 一款8n1 UART发送程序。
12 ; OUT引脚0和side-set引脚0均映射至UART TX引脚。
13
14     pull      side 1 [7] ; 断言停止位，或保持线路处于空闲状态等待
15     set x, 7    side 0 [7] ; 预加载比特计数器，断言起始位持续8个时钟周期
16     bitloop:           ; 该循环将运行8次（8n1 UART）
17         输出引脚, 1          ; 将OSR中1位数据移位至第一个OUT引脚
18         jmp x-- bitloop    [6] ; 每次循环迭代包含8个周期。
```

按当前代码，该程序将：

- 使引脚保持高电平阻塞，直到数据出现（注意即使状态机阻塞，侧置功能仍然生效）
- 断言起始位，持续8个状态机执行周期
- 移位输出8位数据，每位持续8个周期
- 返回空闲线路状态，至少维持8个周期，随后断言下一起始位

若状态机的时钟分频器配置为以目标波特率的8倍运行，则该程序会在数据由软件或系统DMA推送至TX FIFO时，传输格式规范的UART串行帧。若需扩展以支持不同帧长（不同数据位数），可将`set x, 7`替换为`mov x, y`，使`y`寄存器成为每个状态机的UART帧大小配置寄存器。

SDK中的.pio文件也包含此功能，用于配置引脚和状态机，该程序一旦加载至PIO指令存储器：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第24至51行

```
24 static inline void uart_tx_program_init(PIO pio, uint sm, uint offset, uint pin_tx, uint
baud) {
25     // 指示PIO在所选引脚上初始输出高电平，然后通过IO复用器将PIO映射至该引脚。
26     // onto that pin with the IO muxes.
27     pio_sm_set_pins_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
28     pio_sm_set_pindirs_with_mask64(pio, sm, 1ull << pin_tx, 1ull << pin_tx);
29     pio_gpio_init(pio, pin_tx);
30
31     pio_sm_config c = uart_tx_program_get_default_config(offset);
32
33     // 输出向右移动，无自动拉取
34     sm_config_set_out_shift(&c, true, false, 32);
35
36     // 我们将OUT和side-set映射到同一引脚，因为有时
37     // 需要将用户数据（通过OUT）输出到引脚，有时
38     // 需要输出固定值（起始位/停止位）
39     sm_config_set_out_pins(&c, pin_tx, 1);
40     sm_config_set_sideset_pins(&c, pin_tx);
```

```

41
42 // We only need TX, so get an 8-deep FIFO!
43 sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
44
45 // SM transmits 1 bit per 8 execution cycles.
46 float div = (float)clock_get_hz(clk_sys) / (8 * baud);
47 sm_config_set_clkdiv(&c, div);
48
49 pio_sm_init(pio, sm, offset, &c);
50 pio_sm_set_enabled(pio, sm, true);
51 }
```

The state machine is configured to shift right in `out` instructions, because UARTs typically send data LSB-first. Once configured, the state machine will print any characters pushed to the TX FIFO.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio Lines 53 - 55

```

53 static inline void uart_tx_program_putc(PIO pio, uint sm, char c) {
54     pio_sm_put_blocking(pio, sm, (uint32_t)c);
55 }
```

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio Lines 57 - 60

```

57 static inline void uart_tx_program_puts(PIO pio, uint sm, const char *s) {
58     while (*s)
59         uart_tx_program_putc(pio, sm, *s++);
60 }
```

The example program in the SDK will configure one PIO state machine as a UART TX peripheral, and use it to print a message on GPIO 0 at 115200 baud once per second.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.c

```

1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include "pico/stdlib.h"
8 #include "hardware/pio.h"
9 #include "uart_tx.pio.h"
10
11 // We're going to use PIO to print "Hello, world!" on the same GPIO which we
12 // normally attach UART0 to.
13 #define PIO_TX_PIN 0
14
15 // Check the pin is compatible with the platform
16 #error Attempting to use a pin>=32 on a platform that does not support it
17
18 int main() {
19     // This is the same as the default UART baud rate on Pico
20     const uint SERIAL_BAUD = 115200;
21
22     PIO pio;
23     uint sm;
24     uint offset;
25 }
```

```

41
42 // 我们仅需要TX，因此配置一个深度为8的FIFO！
43 sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
44
45 // 状态机每8个执行周期传输1位。
46 float div = (float)clock_get_hz(clk_sys) / (8 * baud);
47 sm_config_set_clkdiv(&c, div);
48
49 pio_sm_init(pio, sm, offset, &c);
50 pio_sm_set_enabled(pio, sm, true);
51 }

```

状态机配置为在 `out` 指令中执行右移，因为UART通常按最低有效位（LSB）优先发送数据。配置完成后，状态机将输出推入TX FIFO的所有字符。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第53至55行

```

53 static inline void uart_tx_program_putc(PIO pio, uint sm, char c) {
54     pio_sm_put_blocking(pio, sm, (uint32_t)c);
55 }

```

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.pio 第57至60行

```

57 static inline void uart_tx_program_puts(PIO pio, uint sm, const char *s) {
58     while (*s)
59         uart_tx_program_putc(pio, sm, *s++);
60 }

```

SDK中的示例程序将配置一个PIO状态机作为UART TX外设，并以115200波特率每秒在GPIO 0上打印消息。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_tx/uart_tx.c

```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3  *
4  * SPDX许可证标识符：BSD-3-Clause
5  */
6
7 #include "pico/stdlib.h"
8 #include "hardware/pio.h"
9 #include "uart_tx.pio.h"
10
11 // 我们将使用PIO在同一GPIO上打印“Hello, world!”，该GPIO通常连接UART0。
12 // 通常将UART0附接至该GPIO。
13 #define PIO_TX_PIN 0
14
15 // 检查引脚是否与平台兼容
16 #error 尝试在不支持的平台上使用引脚编号>=32
17
18 int main() {
19     // 此波特率与Pico上UART的默认波特率相同
20     const uint SERIAL_BAUD = 115200;
21
22     PIO pio;
23     uint sm;
24     uint offset;
25

```

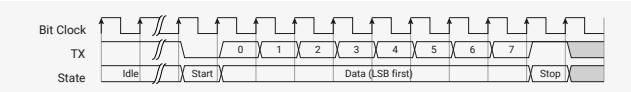
```

26 // This will find a free pio and state machine for our program and load it for us
27 // We use pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range variant)
28 // so we will get a PIO instance suitable for addressing gpios >= 32 if needed and
29 // supported by the hardware
30     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_tx_program, &pio,
31     &sm, &offset, PIO_TX_PIN, 1, true);
32     hard_assert(success);
33
34     uart_tx_program_init(pio, sm, offset, PIO_TX_PIN, SERIAL_BAUD);
35
36     while (true) {
37         uart_tx_program_puts(pio, sm, "Hello, world! (from PIO!)\r\n");
38         sleep_ms(1000);
39     }
40
41 // This will free resources and unload our program
42 pio_remove_program_and_unclaim_sm(&uart_tx_program, pio, sm, offset);
43 }
```

With the two PIO instances on RP2040, this could be extended to 8 additional UART TX interfaces, on 8 different pins, with 8 different baud rates.

3.6.4. UART RX

Recalling Figure 52 showing the format of an 8n1 UART:



We can recover the data by waiting for the start bit, sampling 8 times with the correct timing, and pushing the result to the RX FIFO. Below is possibly the shortest program which can do this:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio Lines 8 - 19

```

8 .program uart_rx_mini
9
10 ; Minimum viable 8n1 UART receiver. Wait for the start bit, then sample 8 bits
11 ; with the correct timing.
12 ; IN pin 0 is mapped to the GPIO used as UART RX.
13 ; Autopush must be enabled, with a threshold of 8.
14
15     wait 0 pin 0          ; Wait for start bit
16     set x, 7 [10]         ; Preload bit counter, delay until eye of first data bit
17     bitloop:              ; Loop 8 times
18     in pins, 1             ; Sample data
19     jmp x-- bitloop [6]   ; Each iteration is 8 cycles
```

This works, but it has some annoying characteristics, like repeatedly outputting **NUL** characters if the line is stuck low. Ideally, we would want to drop data that is not correctly framed by a start and stop bit (and set some sticky flag to indicate this has happened), and pause receiving when the line is stuck low for long periods. We can add these to our program, at the cost of a few more instructions.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio Lines 44 - 63

```

44 .program uart_rx
45
46 ; Slightly more fleshed-out 8n1 UART receiver which handles framing errors and
```

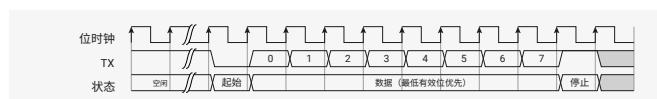
```
26 // 这将为我们的程序寻找一个空闲的 PIO 和状态机，并加载到程序中
27 // 我们使用 pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range 变体)
28 // 因此，如果需要且硬件支持，将获得一个适用于访问 gpio ≥ 32 的 PIO 实例

29     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_tx_program, &pio,
30     &sm, &offset, PIO_TX_PIN, 1, true);
31     hard_assert(success);
32
33     uart_tx_program_init(pio, sm, offset, PIO_TX_PIN, SERIAL_BAUD);
34
35     while (true) {
36         uart_tx_program_puts(pio, sm, "Hello, world! (from PIO!)\r\n");
37         sleep_ms(1000);
38     }
39
40 // 这将释放资源并卸载我们的程序
41     pio_remove_program_and_unclaim_sm(&uart_tx_program, pio, sm, offset);
42 }
```

RP2040上的两个PIO实例可扩展为8个额外的UART TX接口，分布在8个不同引脚，支持8种不同波特率。

3.6.4. UART 接收

回顾图52，展示了8n1UART的格式：



通过等待起始位、以正确时序采样8次并将结果推送至RX FIFO，我们可以恢复数据。以下极可能是实现该功能的最简程序：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第8至19行

```
8 .program uart_rx_mini
9
10 ;最简可用的8n1 UART接收器。等待起始位，然后采样8位
11；确保正确时序。
12 ;IN引脚0映射至用于UART RX的GPIO。
13 ;必须启用自动推送，阈值设置为8。
14
15     wait 0 pin 0          ;等待起始位
16     set x, 7 [10]         ;预加载位计数器，延迟至第一个数据位的采样点
17 bitloop:                  ;循环8次
18     in pins, 1            ;采样数据
19     jmp x-- bitloop [6] ;每次迭代为8个周期
```

该方法可行，但存在一些不便之处，例如当线路持续低电平时，会重复输出 **NUL** 字符。

理想情况下，我们应丢弃未被起始位和停止位正确框定的数据（并设置一个粘性标志以指示该情况），且当线路长时间持续低电平时暂停接收。我们可以将这些功能添加至程序，代价是增加若干指令。

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第44至63行

```
44 .program uart_rx
45
46 ;稍微更完善的8n1 UART接收器，可更稳妥地处理帧错误和
```

```

47 ; break conditions more gracefully.
48 ; IN pin 0 and JMP pin are both mapped to the GPIO used as UART RX.
49
50 start:
51     wait 0 pin 0          ; Stall until start bit is asserted
52     set x, 7    [10]      ; Preload bit counter, then delay until halfway through
53     bitloop:             ; the first data bit (12 cycles incl wait, set).
54     in pins, 1           ; Shift data bit into ISR
55     jmp x-- bitloop [6]  ; Loop 8 times, each loop iteration is 8 cycles
56     jmp pin good_stop   ; Check stop bit (should be high)
57
58     irq 4 rel           ; Either a framing error or a break. Set a sticky flag,
59     wait 1 pin 0         ; and wait for line to return to idle state.
60     jmp start            ; Don't push data if we didn't see good framing.
61
62 good_stop:              ; No delay before returning to start; a little slack is
63     push                 ; important in case the TX clock is slightly too fast.

```

The second example does not use autopush (Section 3.5.4), preferring instead to use an explicit `push` instruction, so that it can condition the push on whether a correct stop bit is seen. The `.pio` file includes a helper function which configures the state machine and connects it to a GPIO with the pull-up enabled:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio Lines 67 - 85

```

67 static inline void uart_rx_program_init(PIO pio, uint sm, uint offset, uint pin, uint baud) {
68     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
69     pio_gpio_init(pio, pin);
70     gpio_pull_up(pin);
71
72     pio_sm_config c = uart_rx_program_get_default_config(offset);
73     sm_config_set_in_pins(&c, pin); // for WAIT, IN
74     sm_config_set_jmp_pin(&c, pin); // for JMP
75     // Shift to right, autopush disabled
76     sm_config_set_in_shift(&c, true, false, 32);
77     // Deeper FIFO as we're not doing any TX
78     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
79     // SM transmits 1 bit per 8 execution cycles.
80     float div = (float)clock_get_hz(clk_sys) / (8 * baud);
81     sm_config_set_clkdiv(&c, div);
82
83     pio_sm_init(pio, sm, offset, &c);
84     pio_sm_set_enabled(pio, sm, true);
85 }

```

To correctly receive data which is sent LSB-first, the ISR is configured to shift to the right. After shifting in 8 bits, this unfortunately leaves our 8 data bits in bits 31:24 of the ISR, with 24 zeroes in the LSBs. One option here is an `in null, 24` instruction to shuffle the ISR contents down to 7:0. Another is to read from the FIFO at an offset of 3 bytes, with an 8-bit read, so that the processor's bus hardware (or the DMA's) picks out the relevant byte for free:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio Lines 87 - 93

```

87 static inline char uart_rx_program_getc(PIO pio, uint sm) {
88     // 8-bit read from the uppermost byte of the FIFO, as data is left-justified
89     io_rw_8 *rxfifo_shift = (io_rw_8*)&pio->rxf[sm] + 3;
90     while (pio_sm_is_rx_fifo_empty(pio, sm))
91         tight_loop_contents();
92     return (char)*rxfifo_shift;
93 }

```

```

47 ;断帧条件。
48 ;IN引脚0和JMP引脚均映射为用作UART RX的GPIO。
49
50 start:
51     等待引脚0为0          ;阻塞，直至检测到起始位
52     设定x为7 [10]；预加载位计数器，随后延时至第一个数据位中点
53 bitloop:           ;（含等待及设定，计12个周期）。
54     从引脚读取，1        ;将数据位移入ISR
55     jmp x--_bitloop [6] ;循环8次，每次循环耗时8周期
56     jmp pin good_stop  ;检测停止位（应为高电平）
57
58     irq 4 rel          ;出现帧错误或断帧时，设置一个粘性标志，
59     等待 1 针 0          ;并等待线路返回空闲状态。
60     jmp start           ;未检测到有效帧时，请勿发送数据。
61
62 good_stop:         ;返回起始点前不延迟；这在
63     发送               ;TX时钟稍快时尤为重要。

```

第二个示例不使用自动发送（第3.5.4节），而是采用显式的 `push` 指令，以便根据是否检测到正确停止位来决定是否发送。该 `.pio` 文件包含一个辅助函数，用以配置状态机并连接启用上拉的 GPIO：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第67至85行

```

67 static inline void uart_rx_program_init(PIO pio, uint sm, uint offset, uint pin, uint baud) {
68     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
69     pio_gpio_init(pio, pin);
70     gpio_pull_up(pin);
71
72     pio_sm_config c = uart_rx_program_get_default_config(offset);
73     sm_config_set_in_pins(&c, pin); // 用于 WAIT 和 IN
74     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
75     // 右移，自动推送功能已禁用
76     sm_config_set_in_shift(&c, true, false, 32);
77     // 由于未进行任何 TX，使用更深的 FIFO
78     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
79     // 状态机每 8 个执行周期传输 1 位
80     float div = (float)clock_get_hz(clk_sys) / (8 * baud);
81     sm_config_set_clkdiv(&c, div);
82
83     pio_sm_init(pio, sm, offset, &c);
84     pio_sm_set_enabled(pio, sm, true);
85 }

```

为正确接收最低有效位优先发送的数据，ISR 配置为右移模式。移入 8 位后，遗憾的是 8 位数据位位于 ISR 的 31:24 位，最低有效位为 24 个零。这里的一个选项是使用一条 `null_24` 指令，将 ISR 内容下移至 7:0 位。另一种方法是从 FIFO 的偏移量 3 字节处进行 8 位读取，以使处理器总线硬件（或 DMA）能够自动选择相关字节：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.pio 第87至93行

```

87 static inline char uart_rx_program_getc(PIO pio, uint sm) {
88     // 从 FIFO 最高字节进行 8 位读取，数据左对齐
89     io_rw_8 *rxfifo_shift = (io_rw_8*)&pio->rxf[sm] + 3;
90     while (pio_sm_is_rx_fifo_empty(pio, sm))
91         tight_loop_contents();
92     return (char)*rxfifo_shift;
93 }

```

An example program shows how this UART RX program can be used to receive characters sent by one of the hardware UARTs on RP2040. A wire must be connected from GPIO4 to GPIO3 for this program to function. To make the wrangling of 3 different serial ports a little easier, this program uses core 1 to print out a string on the test UART (UART 1), and the code running on core 0 will pull out characters from the PIO state machine, and pass them along to the UART used for the debug console (UART 0). Another approach here would be interrupt-based IO, using PIO's FIFO IRQs. If the `SMD_RXNEMPTY` bit is set in the `IRQ0_INTE` register, then PIO will raise its first interrupt request line whenever there is a character in state machine 0's RX FIFO.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.c

```

1 /**
2  * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4  * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stdlib.h"
10 #include "pico/multicore.h"
11 #include "hardware/pio.h"
12 #include "hardware/uart.h"
13 #include "uart_rx.pio.h"
14
15 // This program
16 // - Uses UART1 (the spare UART, by default) to transmit some text
17 // - Uses a PIO state machine to receive that text
18 // - Prints out the received text to the default console (UART0)
19 // This might require some reconfiguration on boards where UART1 is the
20 // default UART.
21
22 #define SERIAL_BAUD PICO_DEFAULT_UART_BAUD_RATE
23 #define HARD_UART_INST uart1
24
25 // You'll need a wire from GPIO4 -> GPIO3
26 #define HARD_UART_TX_PIN 4
27 #define PIO_RX_PIN 3
28
29 // Check the pin is compatible with the platform
30 #error Attempting to use a pin>=32 on a platform that does not support it
31
32 // Ask core 1 to print a string, to make things easier on core 0
33 void core1_main() {
34     const char *s = (const char *) multicore_fifo_pop_blocking();
35     uart_puts(HARD_UART_INST, s);
36 }
37
38 int main() {
39     // Console output (also a UART, yes it's confusing)
40     setup_default_uart();
41     printf("Starting PIO UART RX example\n");
42
43     // Set up the hard UART we're going to use to print characters
44     uart_init(HARD_UART_INST, SERIAL_BAUD);
45     gpio_set_function(HARD_UART_TX_PIN, GPIO_FUNC_UART);
46
47     // Set up the state machine we're going to use to receive them.
48     PIO pio;
49     uint sm;
50     uint offset;
51
52     // This will find a free pio and state machine for our program and load it for us

```

示例程序演示了如何使用该UART RX程序接收RP2040上的硬件UART所发送的字符。程序需将GPIO4连接至GPIO3方可正常运行。为了简化对3个不同串口的管理，本程序使用核心1在测试UART（UART 1）上输出字符串，而核心0运行的代码将从PIO状态机获取字符，并传递给用于调试控制台的UART（UART 0）。另一种方法是基于中断的IO，利用PIO的FIFO中断请求（IRQ）。若IRQ0_INTERRUPT寄存器中的SM0_RXNEEMPTY位被置位，则当状态机0的RX FIFO中有字符时，PIO将触发其第一个中断请求线。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/uart_rx/uart_rx.c

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stdlib.h"
10 #include "pico/multicore.h"
11 #include "hardware/pio.h"
12 #include "hardware/uart.h"
13 #include "uart_rx.pio.h"
14
15 // 本程序
16 // - 使用 UART1 (默认备用 UART) 发送文本
17 // - 使用 PIO 状态机接收该文本
18 // - 将接收到的文本打印至默认控制台 (UART0)
19 // 在某些开发板上, UART1 可能为默认 UART, 可能需重新配置。
20 // default UART.
21
22 #define SERIAL_BAUD PICO_DEFAULT_UART_BAUD_RATE
23 #define HARD_UART_INST uart1
24
25 // 需将 GPIO4 连接至 GPIO3
26 #define HARD_UART_TX_PIN 4
27 #define PIO_RX_PIN 3
28
29 // 请确认该引脚与平台兼容
30 #error 试图使用平台不支持的32号及以上引脚
31
32 // 请求核心1打印字符串, 以简化核心0的操作
33 void core1_main() {
34     const char *s = (const char *) multicore_fifo_pop_blocking();
35     uart_puts(HARD_UART_INST, s);
36 }
37
38 int main() {
39     // 控制台输出 (同样是 UART, 确实容易混淆)
40     setup_default_uart();
41     printf("Starting PIO UART RX example\n");
42
43     // 配置我们将用于打印字符的硬件 UART
44     uart_init(HARD_UART_INST, SERIAL_BAUD);
45     gpio_set_function(HARD_UART_TX_PIN, GPIO_FUNC_UART);
46
47     // 配置我们将用于接收字符的状态机
48     PIO pio;
49     uint sm;
50     uint offset;
51
52     // 该操作将为程序查找并占用空闲的 pio 和状态机, 并进行加载

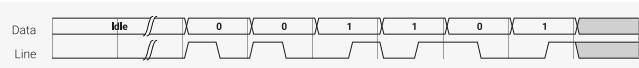
```

```

53     // We use pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range variant)
54     // so we will get a PIO instance suitable for addressing gpios >= 32 if needed and
55     // supported by the hardware
56     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_rx_program, &pio,
57     &sm, &offset, PIO_RX_PIN, 1, true);
58     hard_assert(success);
59
60     uart_rx_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
61     //uart_rx_mini_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
62
63     // Tell core 1 to print some text to uart1 as fast as it can
64     multicore_launch_core1(core1_main);
65     const char *text = "Hello, world from PIO! (Plus 2 UARTs and 2 cores, for complex
66     reasons)\n";
67     multicore_fifo_push_blocking((uint32_t) text);
68
69     // Echo characters received from PIO to the console
70     while (true) {
71         char c = uart_rx_program_getc(pio, sm);
72         putchar(c);
73     }
74
75     // This will free resources and unload our program
76     pio_remove_program_and_unclaim_sm(&uart_rx_program, pio, sm, offset);
77 }
```

3.6.5. Manchester Serial TX and RX

Figure 53. Manchester serial line code. Each data bit is represented by either a high pulse followed by a low pulse (representing a '0' bit) or a low pulse followed by a high pulse (a '1' bit).



Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio Lines 8 - 30

```

8 .program manchester_tx
9 .side_set 1 opt
10
11 ; Transmit one bit every 12 cycles. a '0' is encoded as a high-low sequence
12 ; (each part lasting half a bit period, or 6 cycles) and a '1' is encoded as a
13 ; low-high sequence.
14 ;
15 ; Side-set bit 0 must be mapped to the GPIO used for TX.
16 ; Autopull must be enabled -- this program does not care about the threshold.
17 ; The program starts at the public label 'start'.
18
19 .wrap_target
20 do_1:
21     nop           side 0 [5] ; Low for 6 cycles (5 delay, +1 for nop)
22     jmp get_bit side 1 [3] ; High for 4 cycles. 'get_bit' takes another 2 cycles
23 do_0:
24     nop           side 1 [5] ; Output high for 6 cycles
25     nop           side 0 [3] ; Output low for 4 cycles
26 public start:
27 get_bit:
28     out x, 1          ; Always shift out one bit from OSR to X, so we can
29     jmp !x do_0        ; branch on it. Autopull refills the OSR when empty.
30 .wrap
```

Starting from the label called `start`, this program shifts one data bit at a time into the X register, so that it can branch on

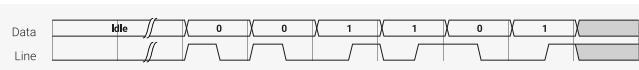
```

53 // 我们使用 pio_claim_free_sm_and_add_program_for_gpio_range (for_gpio_range 变体)
54 // 因此，如果需要且硬件支持，将获得一个适用于访问 gpio ≥ 32 的 PIO 实例

55     bool success = pio_claim_free_sm_and_add_program_for_gpio_range(&uart_rx_program, &pio,
56     &sm, &offset, PIO_RX_PIN, 1, true);
57     hard_assert(success);
58
59     uart_rx_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
60     //uart_rx_mini_program_init(pio, sm, offset, PIO_RX_PIN, SERIAL_BAUD);
61
62     // 指示核心1尽快将文本打印到uart1
63     multicore_launch_core1(core1_main);
64
65     multicore_fifo_push_blocking((uint32_t) text);
66
67     // 将从PIO接收的字符回显至控制台
68     while (true) {
69         char c = uart_rx_program_getc(pio, sm);
70         putchar(c);
71     }
72
73     // 此操作将释放资源并卸载程序
74     pio_remove_program_and_unclaim_sm(&uart_rx_program, pio, sm, offset);
75 }
```

3.6.5. 曼彻斯特编码串行发送与接收

图53. 曼彻斯特码串行线路编码。每个数据位由高脉冲后接低脉冲（表示“0”位）或低脉冲后接高脉冲（表示“1”位）表示。



Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第8至30行

```

8 .program manchester_tx
9 .side_set 1 opt
10
11 ; 每12个周期传输一位。“0”编码为高-低序列
12 ; （每部分持续半个比特周期，即6个周期），“1”编码为
13 ; 低-高序列。
14 ;
15 ; Side-set第0位必须映射至用于TX的GPIO。
16 ; 必须启用自动拉取——本程序不考虑阈值。
17 ; 程序从公共标签“start”处开始。
18
19 .wrap_target
20 do_1:
21     nop           side 0 [5] ; 低电平持续6个周期（包含5个延时周期，外加1个nop周期）
22     jmp get_bit side 1 [3] ; 高电平持续4个周期。'get_bit' 指令额外占用2个周期
23 do_0:
24     nop           side 1 [5] ; 输出高电平持续6个周期
25     nop           side 0 [3] ; 输出低电平持续4个周期
26 public start:
27 get_bit:
28     out x, 1           ; 始终将一个位从OSR移位至X寄存器，以便我们
29     jmp !x do_0          ; 根据该位进行分支。Autopull功能在OSR为空时会自动重新填充。
30 .wrap
```

从名为 `start` 的标签开始，本程序每次将一个数据位移入 X 寄存器，以便根据该位进行跳转。

the value. Depending on the outcome, it uses side-set to drive either a 1-0 or 0-1 sequence onto the chosen GPIO. This program uses autopull ([Section 3.5.4.2](#)) to automatically replenish the OSR from the TX FIFO once a certain amount of data has been shifted out, without interrupting program control flow or timing. This feature is enabled by a helper function in the `.pio` file which configures and starts the state machine:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio Lines 33 - 46

```

33 static inline void manchester_tx_program_init(PIO pio, uint sm, uint offset, uint pin, float
      div) {
34     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
35     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
36     pio_gpio_init(pio, pin);
37
38     pio_sm_config c = manchester_tx_program_get_default_config(offset);
39     sm_config_set_sideset_pins(&c, pin);
40     sm_config_set_out_shift(&c, true, true, 32);
41     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
42     sm_config_set_clkdiv(&c, div);
43     pio_sm_init(pio, sm, offset + manchester_tx_offset_start, &c);
44
45     pio_sm_set_enabled(pio, sm, true);
46 }
```

Another state machine can be programmed to recover the original data from the transmitted signal:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio Lines 49 - 71

```

49 .program manchester_rx
50
51 ; Assumes line is idle low, first bit is 0
52 ; One bit is 12 cycles
53 ; a '0' is encoded as 10
54 ; a '1' is encoded as 01
55 ;
56 ; Both the IN base and the JMP pin mapping must be pointed at the GPIO used for RX.
57 ; Autopush must be enabled.
58 ; Before enabling the SM, it should be placed in a 'wait 1, pin` state, so that
59 ; it will not start sampling until the initial line idle state ends.
60
61 start_of_0:           ; We are 0.25 bits into a 0 - signal is high
62     wait 0 pin 0       ; Wait for the 1->0 transition - at this point we are 0.5 into the bit
63     in y, 1 [8]        ; Emit a 0, sleep 3/4 of a bit
64     jmp pin start_of_0 ; If signal is 1 again, it's another 0 bit, otherwise it's a 1
65
66 .wrap_target
67 start_of_1:           ; We are 0.25 bits into a 1 - signal is 1
68     wait 1 pin 0       ; Wait for the 0->1 transition - at this point we are 0.5 into the bit
69     in x, 1 [8]        ; Emit a 1, sleep 3/4 of a bit
70     jmp pin start_of_0 ; If signal is 0 again, it's another 1 bit otherwise it's a 0
71 .wrap
```

The main complication here is staying aligned to the input transitions, as the transmitter's and receiver's clocks may drift relative to one another. In Manchester code there is always a transition in the centre of the symbol, and based on the initial line state (high or low) we know the direction of this transition, so we can use a `wait` instruction to resynchronise to the line transitions on every data bit.

This program expects the X and Y registers to be initialised with the values 1 and 0 respectively, so that a constant 1 or 0 can be provided to the `in` instruction. The code that configures the state machine initialises these registers by executing some `set` instructions before setting the program running.

该值。根据结果，使用 side-set 在所选 GPIO 上驱动 1-0 或 0-1 序列。该程序采用 autopull（第 3.5.4.2 节）功能，当移出一定量数据后，自动从 TX FIFO 补充 OSR，且不会中断程序控制流或时序。该功能由 .pio 文件中的辅助函数实现，该函数负责配置并启动状态机：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio，第 33 至 46 行

```

33 static inline void manchester_tx_program_init(PIO pio, uint sm, uint offset, uint pin, float
      div) {
34     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
35     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
36     pio_gpio_init(pio, pin);
37
38     pio_sm_config c = manchester_tx_program_get_default_config(offset);
39     sm_config_set_sideset_pins(&c, pin);
40     sm_config_set_out_shift(&c, true, true, 32);
41     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
42     sm_config_set_clkdiv(&c, div);
43     pio_sm_init(pio, sm, offset + manchester_tx_offset_start, &c);
44
45     pio_sm_set_enabled(pio, sm, true);
46 }
```

可编程另一个状态机以从传输信号中恢复原始数据：

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第49至71行

```

49 .program manchester_rx
50
51 ; 假设线路空闲为低电平，第一位为0
52 ; 一位占12个周期
53 ; “0” 编码为 10
54 ; “1” 编码为 01
55 ;
56 ; IN 基址和 JMP 引脚映射必须指向用于接收的 GPIO。
57 ; 必须启用自动推送 (Autopush)。
58 ; 启用状态机前，应将其置于 “wait 1, pin” 状态，以便
59 ; 在初始线路空闲状态结束之前，采样不会开始。
60
61 start_of_0:           ; 我们已进入0信号的0.25位 — 信号为高电平
62     等待引脚0 变为0       ; 等待1→0跳变 — 此时已进入比特的0.5位
63     输入 y, 1 [8]        ; 发出0，休眠3/4位时长
64     jmp pin start_of_0 ; 若信号再次为1，则为另一个0位，否则为1
65
66 .wrap_target
67 start_of_1:           ; 我们已进入1信号的0.25位 — 信号为1
68     等待引脚1 变为0       ; 等待0→1跳变 — 此时已进入比特的0.5位
69     输入 x, 1 [8]        ; 发出1，休眠3/4位时长
70     jmp pin start_of_0 ; 若信号再次为0，则为另一个1位，否则为0
71 .wrap
```

此处的主要难点在于保持对输入转换的同步，因为发射器和接收器的时钟可能会相互漂移。在曼彻斯特码中，符号中间始终存在一次状态转换，且基于初始线路状态（高电平或低电平），我们可确定该转换的方向，因此可利用 wait 指令在每个数据位上重新同步线路转换。

本程序要求 X 和 Y 寄存器分别初始化为 1 和 0，以便向 in 指令提供恒定的 1 或 0。配置状态机的代码通过执行若干 set 指令初始化这些寄存器，随后启动程序运行。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio Lines 74 - 94

```

74 static inline void manchester_rx_program_init(PIO pio, uint sm, uint offset, uint pin, float
    div) {
75     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
76     pio_gpio_init(pio, pin);
77
78     pio_sm_config c = manchester_rx_program_get_default_config(offset);
79     sm_config_set_in_pins(&c, pin); // for WAIT
80     sm_config_set_jmp_pin(&c, pin); // for JMP
81     sm_config_set_in_shift(&c, true, true, 32);
82     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
83     sm_config_set_clkdiv(&c, div);
84     pio_sm_init(pio, sm, offset, &c);
85
86     // X and Y are set to 0 and 1, to conveniently emit these to ISR/FIFO.
87     pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
88     pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
89     // Assume line is idle low, and first transmitted bit is 0. Put SM in a
90     // wait state before enabling. RX will begin once the first 0 symbol is
91     // detected.
92     pio_sm_exec(pio, sm, pio_encode_wait_pin(1, 0) | pio_encode_delay(2));
93     pio_sm_set_enabled(pio, sm, true);
94 }
```

The example C program in the SDK will transmit Manchester serial data from GPIO2 to GPIO3 at approximately 10Mbps (assuming a system clock of 125MHz).

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.c Lines 20 - 43

```

20 int main() {
21     stdio_init_all();
22
23     PIO pio = pio0;
24     uint sm_tx = 0;
25     uint sm_rx = 1;
26
27     uint offset_tx = pio_add_program(pio, &manchester_tx_program);
28     uint offset_rx = pio_add_program(pio, &manchester_rx_program);
29     printf("Transmit program loaded at %d\n", offset_tx);
30     printf("Receive program loaded at %d\n", offset_rx);
31
32     manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 1.f);
33     manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 1.f);
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }
```

3.6.6. Differential Manchester (BMC) TX and RX

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.pio 第74至94行

```

74 static inline void manchester_rx_program_init(PIO pio, uint sm, uint offset, uint pin, float
    div) {
75     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
76     pio_gpio_init(pio, pin);
77
78     pio_sm_config c = manchester_rx_program_get_default_config(offset);
79     sm_config_set_in_pins(&c, pin); // 用于 WAIT
80     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
81     sm_config_set_in_shift(&c, true, true, 32);
82     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
83     sm_config_set_clkdiv(&c, div);
84     pio_sm_init(pio, sm, offset, &c);
85
86     // 将 X 和 Y 分别设置为 0 和 1，以便方便地发送到 ISR/FIFO。
87     pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
88     pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
89     // 假设线路空闲时为低电平，且首个发送的位为 0。启用前将状态机置于
90     // 等待状态。RX 将在检测到第一个 0 符号后启动。
91     // detected.
92     pio_sm_exec(pio, sm, pio_encode_wait_pin(1, 0) | pio_encode_delay(2));
93     pio_sm_set_enabled(pio, sm, true);
94 }
```

SDK中的示例C程序将以约10Mbps的速度从GPIO2向GPIO3传输曼彻斯特码串行数据
(假设系统时钟频率为125MHz)。

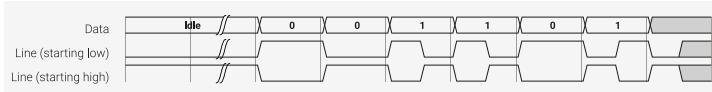
Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/manchester_encoding/manchester_encoding.c 第20至43行

```

20 int main() {
21     stdio_init_all();
22
23     PIO pio = pio0;
24     uint sm_tx = 0;
25     uint sm_rx = 1;
26
27     uint offset_tx = pio_add_program(pio, &manchester_tx_program);
28     uint offset_rx = pio_add_program(pio, &manchester_rx_program);
29     printf("传输程序加载于%d\n", offset_tx);
30     printf("接收程序加载于%d\n", offset_rx);
31
32     manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 1.f);
33     manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 1.f);
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }
```

3.6.6. 差分曼彻斯特编码（BMC）发送与接收

Figure 54. Differential Manchester serial line code, also known as biphase mark code (BMC). The line transitions at the start of every bit period. The presence of a transition in the centre of the bit period signifies a 1 data bit, and the absence, a 0 bit. These encoding rules are the same whether the line has an initial high or low state.



The transmit program is similar to the Manchester example: it repeatedly shifts a bit from the OSR into X (relying on autopull to refill the OSR in the background), branches, and drives a GPIO up and down based on the value of this bit. The added complication is that the pattern we drive onto the pin depends not just on the value of the data bit, as with vanilla Manchester encoding, but also on the state the line was left in at the end of the last bit period. This is illustrated in [Figure 54](#), where the pattern is inverted if the line is initially high. To cope with this, there are two copies of the test-and-drive code, one for each initial line state, and these are linked together in the correct order by a sequence of jumps.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio Lines 8 - 35

```

8 .program differential_manchester_tx
9 .side_set 1 opt
10
11 ; Transmit one bit every 16 cycles. In each bit period:
12 ; - A '0' is encoded as a transition at the start of the bit period
13 ; - A '1' is encoded as a transition at the start *and* in the middle
14 ;
15 ; Side-set bit 0 must be mapped to the data output pin.
16 ; Autopull must be enabled.
17
18 public start:
19 initial_high:
20     out x, 1                      ; Start of bit period: always assert transition
21     jmp !x high_0    side 1 [6] ; Test the data bit we just shifted out of OSR
22 high_1:
23     nop
24     jmp initial_high  side 0 [6] ; For `1` bits, also transition in the middle
25 high_0:
26     jmp initial_low       [7] ; Otherwise, the line is stable in the middle
27
28 initial_low:
29     out x, 1                      ; Always shift 1 bit from OSR to X so we can
30     jmp !x low_0     side 0 [6] ; branch on it. Autopull refills OSR for us.
31 low_1:
32     nop
33     jmp initial_low  side 1 [6] ; If there are two transitions, return to
34 low_0:
35     jmp initial_high       [7] ; the initial line state is flipped!

```

The `.pio` file also includes a helper function to initialise a state machine for differential Manchester TX, and connect it to a chosen GPIO. We arbitrarily choose a 32-bit frame size and LSB-first serialisation (`shift_to_right` is true in `sm_config_set_out_shift`), but as the program operates on one bit at a time, we could change this by reconfiguring the state machine.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio Lines 38 - 53

```

38 static inline void differential_manchester_tx_program_init(PIO pio, uint sm, uint offset,
    uint pin, float div) {
39     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
40     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
41     pio_gpio_init(pio, pin);
42
43     pio_sm_config c = differential_manchester_tx_program_get_default_config(offset);
44     sm_config_set_sideset_pins(&c, pin);
45     sm_config_set_out_shift(&c, true, true, 32);
46     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
47     sm_config_set_clkdiv(&c, div);

```

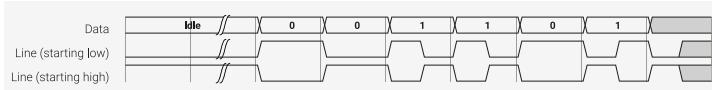
图54。差分曼彻斯特特码，又称双相标记码（BMC）。线路在每个位周期开始时发生跳变。

位周期中间的

跳变表示数据位 1

，

以及缺失，即一个 0 比特。这些编码规则无论线路的初始状态是高电平还是低电平均相



发送程序类似于曼彻斯特编码示例：它不断地将一个比特从OSR移位至X（依赖自动拉取机制在后台重新填充OSR），根据该比特的值分支，并驱动GPIO引脚的电平升降。
增加的复杂性在于，我们驱动至引脚的信号模式不仅取决于数据比特的值（如传统曼彻斯特编码），还取决于上一个比特周期结束时线路所保持的电平状态。如图54所示，当线路初始为高电平时，信号模式将被反转。为应对这一情况，存在两份测试与驱动代码，分别对应两种初始线路状态，并通过一系列跳转指令以正确顺序将它们连接起来。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 8-35行

```

8 .program differential_manchester_tx
9 .side_set 1 opt
10
11 ; 每16个周期传输1个位。在每个位周期内：
12 ; - '0' 编码为位周期开始时的跳变
13 ; - '1' 编码为位周期开始和中间均有跳变
14 ;
15 ; side-set位0必须映射至数据输出引脚。
16 ; 必须启用autopull功能。
17
18 public start:
19 initial_high:
20     out x, 1           ; 位周期开始时：始终产生跳变
21     jmp !x high_0     side 1 [6] ; 测试刚从OSR移出的数据位
22 high_1:
23     nop
24     jmp initial_high side 0 [6] ; 对于 '1' 位，位周期中间也需跳变
25 high_0:
26     jmp initial_low      [7] ; 否则，该线路在中间保持稳定
27
28 initial_low:
29     out x, 1           ; 始终将1位从OSR移位至X，因此我们可以
30     jmp !x low_0        side 0 [6]; 据此分支。Autopull为我们自动补充OSR。
31 low_1:
32     nop
33     jmp initial_low    side 1 [6]; 如果有两次跳变，则返回
34 low_0:
35     jmp initial_high    [7] ; 初始线路状态已翻转!

```

该 .pio 文件还包含一个辅助函数，用于初始化差分曼彻斯特发送（TX）状态机，并将其连接至指定GPIO。我们任意选择了32位帧长和最低有效位优先的序列化（`shift_to_right`在`sm_config_set_out_shift`中设为true），但由于程序一次操作一位，我们可通过重新配置状态机予以修改。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第38至53行

```

38 static inline void differential_manchester_tx_program_init(PIO pio, uint sm, uint offset,
      uint pin, float div) {
39     pio_sm_set_pins_with_mask(pio, sm, 0, 1u << pin);
40     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
41     pio_gpio_init(pio, pin);
42
43     pio_sm_config c = differential_manchester_tx_program_get_default_config(offset);
44     sm_config_set_sideset_pins(&c, pin);
45     sm_config_set_out_shift(&c, true, true, 32);
46     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
47     sm_config_set_clkdiv(&c, div);

```

```

48     pio_sm_init(pio, sm, offset + differential_manchester_tx_offset_start, &c);
49
50     // Execute a blocking pull so that we maintain the initial line state until data is
51     // available
51     pio_sm_exec(pio, sm, pio_encode_pull(false, true));
52     pio_sm_set_enabled(pio, sm, true);
53 }
```

The RX program uses the following strategy:

- Wait until the initial transition at the start of the bit period, so we stay aligned to the transmit clock
- Then wait 3/4 of the configured bit period, so that we are centred on the second half-bit-period (see [Figure 54](#))
- Sample the line at this point to determine whether there are one or two transitions in this bit period
- Repeat

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio Lines 55 - 85

```

55 .program differential_manchester_rx
56
57 ; Assumes line is idle low
58 ; One bit is 16 cycles. In each bit period:
59 ; - A '0' is encoded as a transition at time 0
60 ; - A '1' is encoded as a transition at time 0 and a transition at time T/2
61 ;
62 ; The IN mapping and the JMP pin select must both be mapped to the GPIO used for
63 ; RX data. Autopush must be enabled.
64
65 public start:
66 initial_high:           ; Find rising edge at start of bit period
67     wait 1 pin, 0 [11] ; Delay to eye of second half-period (i.e 3/4 of way
68     jmp pin high_0      ; through bit) and branch on RX pin high/low.
69 high_1:
70     in x, 1             ; Second transition detected (a '1' data symbol)
71     jmp initial_high
72 high_0:
73     in y, 1 [1]         ; Line still high, no centre transition (data is '0')
74     ; Fall-through
75
76 .wrap_target
77 initial_low:           ; Find falling edge at start of bit period
78     wait 0 pin, 0 [11] ; Delay to eye of second half-period
79     jmp pin low_1
80 low_0:
81     in y, 1             ; Line still low, no centre transition (data is '0')
82     jmp initial_high
83 low_1:                  ; Second transition detected (data is '1')
84     in x, 1 [1]
85 .wrap
```

This code assumes that X and Y have the values 1 and 0, respectively. This is arranged for by the included C helper function:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio Lines 88 - 104

```

88 static inline void differential_manchester_rx_program_init(PIO pio, uint sm, uint offset,
89               uint pin, float div) {
90     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
91     pio_gpio_init(pio, pin);
```

```

48     pio_sm_init(pio, sm, offset + differential_manchester_tx_offset_start, &c);
49
50 // 执行阻塞拉取，以保持初始线路状态直到数据可用

51     pio_sm_exec(pio, sm, pio_encode_pull(false, true));
52     pio_sm_set_enabled(pio, sm, true);
53 }

```

接收程序采用以下策略：

- 等待直到位周期开始时的初始跳变，确保与发送时钟保持同步
- 然后等待配置的位周期的3/4，以使采样点位于半位周期的后半部分中心（参见图54）
- 在该点采样线路，以确定该比特周期内是否存在一次或两次跳变
- 重复

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第55至85行

```

55 .program differential_manchester_rx
56
57 ; 假定线路空闲时为低电平
58 ; 一个比特周期为 16 个周期。在每个比特周期内：
59 ; - “0”的编码为在时间 0 发生一次跳变
60 ; - “1”的编码为在时间 0 及时间 T/2 各发生一次跳变
61 ;
62 ; IN 映射和 JMP 引脚选择均须映射至所用 GPIO
63 ; RX 数据。必须启用自动推送 (Autopush)。
64
65 public start:
66 initial_high:           ; 在比特周期开始处检测上升沿
67     wait 1 pin, 0 [11] ; 延时至第二半周期的眼位 (即 3/4 处)
68     jmp pin high_0      ; 通过位) 并根据 RX 引脚的高/低电平进行跳转。
69 high_1:
70     in x, 1             ; 检测到第二次跳变 (`1` 数据符号)
71     jmp initial_high
72 high_0:
73     in y, 1 [1]         ; 线路仍为高电平，未检测到中间跳变 (数据为 `0`)
74     ; 直接落入
75
76 .wrap_target
77 initial_low:            ; 查找位周期起始处的下降沿
78     wait 0 pin, 0 [11] ; 延迟至半周期后半部分的采样时刻
79     jmp pin low_1
80 low_0:
81     in y, 1             ; 线路仍为低电平，未检测到中间跳变 (数据为 `0`)
82     jmp initial_high
83 low_1:                  ; 检测到第二次跳变 (数据为 `1`)
84     in x, 1 [1]
85 .wrap

```

该代码假设X和Y的值分别为1和0。此功能由所包含的C辅助函数实现：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.pio 第88至104行

```

88 static inline void differential_manchester_rx_program_init(PIO pio, uint sm, uint offset,
89               uint pin, float div) {
90     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, false);
91     pio_gpio_init(pio, pin);
92

```

```

92     pio_sm_config c = differential_manchester_rx_program_get_default_config(offset);
93     sm_config_set_in_pins(&c, pin); // for WAIT
94     sm_config_set_jmp_pin(&c, pin); // for JMP
95     sm_config_set_in_shift(&c, true, true, 32);
96     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
97     sm_config_set_clkdiv(&c, div);
98     pio_sm_init(pio, sm, offset, &c);
99
100    // X and Y are set to 0 and 1, to conveniently emit these to ISR/FIFO.
101   pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
102   pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
103   pio_sm_set_enabled(pio, sm, true);
104 }
```

All the pieces now exist to loopback some serial data over a wire between two GPIOs.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.c

```

1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stlolib.h"
10 #include "hardware/pio.h"
11 #include "differential_manchester.pio.h"
12
13 // Differential serial transmit/receive example
14 // Need to connect a wire from GPIO2 -> GPIO3
15
16 const uint pin_tx = 2;
17 const uint pin_rx = 3;
18
19 int main() {
20     stdio_init_all();
21
22     PIO pio = pio0;
23     uint sm_tx = 0;
24     uint sm_rx = 1;
25
26     uint offset_tx = pio_add_program(pio, &differential_manchester_tx_program);
27     uint offset_rx = pio_add_program(pio, &differential_manchester_rx_program);
28     printf("Transmit program loaded at %d\n", offset_tx);
29     printf("Receive program loaded at %d\n", offset_rx);
30
31     // Configure state machines, set bit rate at 5 Mbps
32     differential_manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 125.f / (16 * 5));
33     differential_manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 125.f / (16 * 5));
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }
```

```

92     pio_sm_config c = differential_manchester_rx_program_get_default_config(offset);
93     sm_config_set_in_pins(&c, pin); // 用于 WAIT
94     sm_config_set_jmp_pin(&c, pin); // 用于 JMP
95     sm_config_set_in_shift(&c, true, true, 32);
96     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
97     sm_config_set_clkdiv(&c, div);
98     pio_sm_init(pio, sm, offset, &c);
99
100    // X和Y被设定为0和1，便于向ISR/FIFO发送。
101    pio_sm_exec(pio, sm, pio_encode_set(pio_x, 1));
102    pio_sm_exec(pio, sm, pio_encode_set(pio_y, 0));
103    pio_sm_set_enabled(pio, sm, true);
104 }

```

所有组件均已具备，现可通过两路 GPIO 之间的一根导线实现串行数据的回环传输。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/differential_manchester/differential_manchester.c

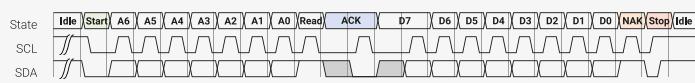
```

1 /**
2  * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4  * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8
9 #include "pico/stlolib.h"
10 #include "hardware/pio.h"
11 #include "differential_manchester.pio.h"
12
13 // 差分串行收发示例
14 // 需将导线连接 GPIO2 至 GPIO3
15
16 const uint pin_tx = 2;
17 const uint pin_rx = 3;
18
19 int main() {
20     stdio_init_all();
21
22     PIO pio = pio0;
23     uint sm_tx = 0;
24     uint sm_rx = 1;
25
26     uint offset_tx = pio_add_program(pio, &differential_manchester_tx_program);
27     uint offset_rx = pio_add_program(pio, &differential_manchester_rx_program);
28     printf("发送程序加载于 %d\n", offset_tx);
29     printf("接收程序加载于 %d\n", offset_rx);
30
31     // 配置状态机，设置比特率为 5 Mbps
32     differential_manchester_tx_program_init(pio, sm_tx, offset_tx, pin_tx, 125.f / (16 * 5));
33     differential_manchester_rx_program_init(pio, sm_rx, offset_rx, pin_rx, 125.f / (16 * 5));
34
35     pio_sm_set_enabled(pio, sm_tx, false);
36     pio_sm_put_blocking(pio, sm_tx, 0);
37     pio_sm_put_blocking(pio, sm_tx, 0x0ff0a55a);
38     pio_sm_put_blocking(pio, sm_tx, 0x12345678);
39     pio_sm_set_enabled(pio, sm_tx, true);
40
41     for (int i = 0; i < 3; ++i)
42         printf("%08x\n", pio_sm_get_blocking(pio, sm_rx));
43 }

```

3.6.7. I2C

Figure 55. A 1-byte I2C read transfer. In the idle state, both lines float high. The initiator drives SDA low (a Start condition), followed by 7 address bits A6-A0, and a direction bit (Read/nWrite). The target drives SDA low to acknowledge the address (ACK). Data bytes follow. The target serialises data on SDA, clocked out by SCL. Every 9th clock, the initiator pulls SDA low to acknowledge the data, except on the last byte, where it leaves the line high (NAK). Releasing SDA whilst SCL is high is a Stop condition, returning the bus to idle.



I2C is an ubiquitous serial bus first described in the Dead Sea Scrolls, and later used by Philips Semiconductor. Two wires with pull-up resistors form an open-drain bus, and multiple agents address and signal one another over this bus by driving the bus lines low, or releasing them to be pulled high. It has a number of unusual attributes:

- SCL can be held low at any time, for any duration, by any member of the bus (not necessarily the target or initiator of the transfer). This is known as clock stretching. The bus does not advance until all drivers release the clock.
- Members of the bus can be a target of one transfer and initiate other transfers (the master/slave roles are not fixed). However this is poorly supported by most I2C hardware.
- SCL is not an edge-sensitive clock, rather SDA must be valid the entire time SCL is high
- In spite of the transparency of SDA against SCL, transitions of SDA whilst SCL is high are used to mark beginning and end of transfers (Start/Stop), or a new address phase within one (Restart)

The PIO program listed below handles serialisation, clock stretching, and checking of ACKs in the initiator role. It provides a mechanism for escaping PIO instructions in the FIFO datastream, to issue Start/Stop/Restart sequences at appropriate times. Provided no unexpected NAKs are received, this can perform long sequences of I2C transfers from a DMA buffer, without processor intervention.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> Lines 8 - 73

```

8 .program i2c
9 .side_set 1 opt pindirs
10
11 ; TX Encoding:
12 ; | 15:10 | 9      | 8:1   | 0    |
13 ; | Instr | Final | Data  | NAK |
14 ;
15 ; If Instr has a value n > 0, then this FIFO word has no
16 ; data payload, and the next n + 1 words will be executed as instructions.
17 ; Otherwise, shift out the 8 data bits, followed by the ACK bit.
18 ;
19 ; The Instr mechanism allows stop/start/repstart sequences to be programmed
20 ; by the processor, and then carried out by the state machine at defined points
21 ; in the datastream.
22 ;
23 ; The "Final" field should be set for the final byte in a transfer.
24 ; This tells the state machine to ignore a NAK: if this field is not
25 ; set, then any NAK will cause the state machine to halt and interrupt.
26 ;
27 ; Autopull should be enabled, with a threshold of 16.
28 ; Autopush should be enabled, with a threshold of 8.
29 ; The TX FIFO should be accessed with halfword writes, to ensure
30 ; the data is immediately available in the OSR.
31 ;
32 ; Pin mapping:
33 ; - Input pin 0 is SDA, 1 is SCL (if clock stretching used)
34 ; - Jump pin is SDA
35 ; - Side-set pin 0 is SCL
36 ; - Set pin 0 is SDA
37 ; - OUT pin 0 is SDA
38 ; - SCL must be SDA + 1 (for wait mapping)
39 ;
40 ; The OE outputs should be inverted in the system IO controls!
41 ; (It's possible for the inversion to be done in this program,
42 ; but costs 2 instructions: 1 for inversion, and one to cope

```

3.6.7. I2C

图 55。一次 1 字节的 I2C 读取传输。
空闲状态下，两条线路均处于高电平漂浮状态。

主设备将 SDA 拉低（起始条件），随后发送 7 位地址 A6-A0 及一个方向位（读/写）。

目标设备将 SDA 拉低以确认地址（ACK），随后传输数据字节。目标设备沿 SDA 线串行传输数据，由 SCL 时钟控制输出。

每第 9 个时钟周期，主设备将 SDA 拉低以确认数据，最后一个字节除外，主设备保持线路为高电平（NAK 当 SCL 为高电平时释放 SDA 即为停止条件，令总线返回空闲状态。



I2C 是一种广泛使用的串行总线，最早记载于《死海古卷》，后由飞利浦半导体采用。两根线配合上拉电阻构成开漏总线，多个设备通过将总线拉低或释放线路让其被上拉到高电平，进行地址定位及信号通信。其具备若干特殊属性：

- SCL 信号线可由总线上的任何成员在任何时间、任意时长拉低（不必是传输的目标或发起方）。此现象称为时钟延长。在所有驱动器释放时钟之前，总线不会前进。
- 总线成员既可作为某次传输的目标，也可发起其他传输（主从角色不固定）。然而，大多数 I2C 硬件对此支持较差。
- SCL 不是边沿敏感的时钟线，而 SDA 必须在 SCL 处于高电平时始终有效。
- 尽管 SDA 对 SCL 具有透明性，但在 SCL 高电平期间 SDA 的跳变用于标记传输的开始和结束（起始/停止），或单次传输中的新地址阶段（重复启动）。

以下 PIO 程序在发起方角色中负责处理串行化、时钟延长及 ACK 检测。该程序提供一种机制，可在 FIFO 数据流中对 PIO 指令进行转义，以在适当时机发出起始/停止/重复启动序列。只要未收到意外的 NAK，该机制即可从 DMA 缓冲区执行长序列的 I2C 传输，无需处理器干预。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第 8 行至第 73 行

```

8 .program i2c
9 .side_set 1 opt pindirs
10
11 ; 发送编码：
12 ; | 15:10 | 9      | 8:1   | 0    |
13 ; | Instr | 结束 | 数据 | NAK |
14 ;
15 ; 若 Instr 的值 n > 0，则该 FIFO 字无
16 ; 数据负载，接下来的 n + 1 个字将作为指令执行。
17 ; 否则，先移出 8 位数据位，随后是 ACK 位。
18 ;
19 ; Instr 机制允许由处理器编程停止/启动/重复启动序列，
20 ; 并由状态机在数据流中的指定点执行，
21 ; in the datastream.
22 ;
23 ; "Final" 字段应在传输的最后一个字节中设置。
24 ; 此设置指示状态机忽略 NAK：若此字段未设置，
25 ; 则任何 NAK 将导致状态机停止并产生中断。
26 ;
27 ; 应启用自动拉取，阈值设为 16。
28 ; 应启用自动推送，阈值设为 8。
29 ; 应采用半字写入方式访问 TX FIFO，以确保
30 ; 数据能够即时在 OSR 中可用。
31 ;
32 ; 引脚映射：
33 ; - 输入引脚 0 为 SDA，1 为 SCL（若使用时钟拉伸）
34 ; - 跳转引脚为 SDA
35 ; - 侧设引脚 0 为 SCL
36 ; - 置位引脚 0 为 SDA
37 ; - 输出引脚 0 为 SDA
38 ; - SCL 必须为 SDA + 1（用于等待映射）
39 ;
40 ; 系统 IO 控制中 OE 输出应取反！
41 ; （该程序中可实现反转，
42 ; 但需消耗两条指令：一条用于反转，另一条用于相应处理）

```

```

43 ; with the side effect of the MOV on TX shift counter.)
44
45 do_nack:
46     jmp y-- entry_point      ; Continue if NAK was expected
47     irq wait 0 rel          ; Otherwise stop, ask for help
48
49 do_byte:
50     set x, 7                ; Loop 8 times
51 bitloop:
52     out pindirs, 1          [7] ; Serialise write data (all-ones if reading)
53     nop                      side 1 [2] ; SCL rising edge
54     wait 1 pin, 1            [4] ; Allow clock to be stretched
55     in pins, 1               [7] ; Sample read data in middle of SCL pulse
56     jmp x-- bitloop side 0 [7] ; SCL falling edge
57
58     ; Handle ACK pulse
59     out pindirs, 1          [7] ; On reads, we provide the ACK.
60     nop                      side 1 [7] ; SCL rising edge
61     wait 1 pin, 1            [7] ; Allow clock to be stretched
62     jmp pin do_nack side 0 [2] ; Test SDA for ACK/NAK, fall through if ACK
63
64 public entry_point:
65 .wrap_target
66     out x, 6                ; Unpack Instr count
67     out y, 1                ; Unpack the NAK ignore bit
68     jmp !x do_byte          ; Instr == 0, this is a data record.
69     out null, 32             ; Instr > 0, remainder of this OSR is invalid
70 do_exec:
71     out exec, 16             ; Execute one instruction per FIFO word
72     jmp x-- do_exec         ; Repeat n + 1 times
73 .wrap

```

The IO mapping required by the I2C program is quite complex, due to the different ways that the two serial lines must be driven and sampled. One interesting feature is that state machine must drive the output enable high when the output is low, since the bus is open-drain, so the sense of the data is inverted. This could be handled in the PIO program (e.g. `mov osr`, `~osr`), but instead we can use the IO controls on RP2040 to perform this inversion in the GPIO muxes, saving an instruction.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> Lines 81 - 121

```

81 static inline void i2c_program_init(PIO pio, uint sm, uint offset, uint pin_sda, uint
82     pin_scl) {
83     assert(pin_scl == pin_sda + 1);
84     pio_sm_config c = i2c_program_get_default_config(offset);
85
86     // IO mapping
87     sm_config_set_out_pins(&c, pin_sda, 1);
88     sm_config_set_set_pins(&c, pin_sda, 1);
89     sm_config_set_in_pins(&c, pin_sda);
90     sm_config_set_sideset_pins(&c, pin_scl);
91     sm_config_set_jmp_pin(&c, pin_sda);
92
93     sm_config_set_out_shift(&c, false, true, 16);
94     sm_config_set_in_shift(&c, false, true, 8);
95
96     float div = (float)clock_get_hz(clk_sys) / (32 * 100000);
97     sm_config_set_clkdiv(&c, div);
98
99     // Try to avoid glitching the bus while connecting the IOs. Get things set
100    // up so that pin is driven down when PIO asserts OE low, and pulled up
101    // otherwise.

```

```

43 ; 伴随对TX移位计数器执行MOV的副作用。
44
45 do_nack:
46     jmp y-- entry_point          ; 如果预期接收NAK，则继续
47     irq wait 0 rel              ; 否则停止，寻求协助
48
49 do_byte:
50     set x, 7                   ; 循环8次
51 bitloop:
52     out pindirs, 1             [7] ; 串行写入数据（读取时全为1）
53     nop                      side 1 [2]; SCL上升沿
54     wait 1 pin, 1              [4] ; 允许时钟拉伸
55     in pins, 1                [7] ; 在SCL脉冲中间采样读取数据
56     jmp x-- bitloop side 0 [7]; SCL下降沿
57
58 ; 处理ACK脉冲
59     out pindirs, 1             [7] ; 读取时，我们提供ACK。
60     nop                      side 1 [7]; SCL上升沿
61     wait 1 pin, 1              [7] ; 允许时钟拉伸
62     jmp pindir do_nack side 0 [2]; 测试SDA是否为ACK/NAK，若为ACK则继续执行
63
64 公共入口点：
65 .wrap_target
66     out x, 6                  ; 解包指令计数
67     out y, 1                  ; 解包NAK忽略位
68     jmp !x do_byte           ; 指令 == 0，此为数据记录。
69     out null, 32              ; 指令 > 0，此OSR剩余部分无效
70 do_exec:
71     out exec, 16              ; 每个FIFO字执行一条指令
72     jmp x-- do_exec          ; 重复n+1次
73 .wrap

```

I2C程序所需的IO映射较为复杂，原因在于两条串行线必须以不同方式进行驱动和采样。一个有趣的特性是，状态机必须在输出为低电平时将输出使能置高，因为总线为开漏结构，数据的逻辑含义被反转。该反转本可在PIO程序中处理（例如`mov osr, ~osr`），但也可利用RP2040的IO控制，在GPIO多路复用器中完成，从而节省一条指令。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第81至121行

```

81 static inline void i2c_program_init(PIO pio, uint sm, uint offset, uint pin_sda, uint
82     pin_scl) {
83     assert(pin_scl == pin_sda + 1);
84     pio_sm_config c = i2c_program_get_default_config(offset);
85
86     // IO映射
87     sm_config_set_out_pins(&c, pin_sda, 1);
88     sm_config_set_set_pins(&c, pin_sda, 1);
89     sm_config_set_in_pins(&c, pin_sda);
90     sm_config_set_sideset_pins(&c, pin_scl);
91     sm_config_set_jmp_pin(&c, pin_sda);
92
93     sm_config_set_out_shift(&c, false, true, 16);
94     sm_config_set_in_shift(&c, false, true, 8);
95
96     float div = (float)clock_get_hz(clk_sys) / (32 * 100000);
97     sm_config_set_clkdiv(&c, div);
98
99     // 尝试避免在连接IO时对总线引起毛刺。进行配置
100    // 使当PIO断言OE低电平时管脚被驱动为低电平，并且被上拉
101    // 否则。

```

```

101    gpio_pull_up(pin_scl);
102    gpio_pull_up(pin_sda);
103    uint32_t both_pins = (1u << pin_sda) | (1u << pin_scl);
104    pio_sm_set_pins_with_mask(pio, sm, both_pins, both_pins);
105    pio_sm_set_pindirs_with_mask(pio, sm, both_pins, both_pins);
106    pio_gpio_init(pio, pin_sda);
107    gpio_set_oeover(pin_sda, GPIO_OVERRIDE_INVERT);
108    pio_gpio_init(pio, pin_scl);
109    gpio_set_oeover(pin_scl, GPIO_OVERRIDE_INVERT);
110    pio_sm_set_pins_with_mask(pio, sm, 0, both_pins);
111
112    // Clear IRQ flag before starting, and make sure flag doesn't actually
113    // assert a system-level interrupt (we're using it as a status flag)
114    pio_set_irq0_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
115        sm), false);
115    pio_set_irq1_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
116        sm), false);
117    pio_interrupt_clear(pio, sm);
118
119    // Configure and start SM
120    pio_sm_init(pio, sm, offset + i2c_offset_entry_point, &c);
121    pio_sm_set_enabled(pio, sm, true);
122 }

```

We can also use the PIO assembler to generate a table of instructions for passing through the FIFO, for Start/Stop/Restart conditions.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> Lines 126 - 136

```

126 .program set_scl_sda
127 .side_set 1 opt
128
129 ; Assemble a table of instructions which software can select from, and pass
130 ; into the FIFO, to issue START/STOP/RSTART. This isn't intended to be run as
131 ; a complete program.
132
133     set pindirs, 0 side 0 [7] ; SCL = 0, SDA = 0
134     set pindirs, 1 side 0 [7] ; SCL = 0, SDA = 1
135     set pindirs, 0 side 1 [7] ; SCL = 1, SDA = 0
136     set pindirs, 1 side 1 [7] ; SCL = 1, SDA = 1

```

The example code does blocking software IO on the state machine's FIFOs, to avoid the extra complexity of setting up the system DMA. For example, an I2C start condition is enqueued like so:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c Lines 69 - 73

```

69 void pio_i2c_start(PIO pio, uint sm) {
70     pio_i2c_put_or_err(pio, sm, 1u << PIO_I2C_ICOUNT_LSB); // Escape code for 2 instruction
71     sequence
72     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC1_SD0]); // We are
73     already in idle state, just pull SDA low
74     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC0_SD0]); // Also
75     pull clock low so we can present data
76 }

```

Because I2C can go wrong at so many points, we need to be able to check the error flag asserted by the state machine, clear the halt and restart it, before asserting a Stop condition and releasing the bus.

```

101    gpio_pull_up(pin_scl);
102    gpio_pull_up(pin_sda);
103    uint32_t both_pins = (1u << pin_sda) | (1u << pin_scl);
104    pio_sm_set_pins_with_mask(pio, sm, both_pins, both_pins);
105    pio_sm_set_pindirs_with_mask(pio, sm, both_pins, both_pins);
106    pio_gpio_init(pio, pin_sda);
107    gpio_set_oeover(pin_sda, GPIO_OVERRIDE_INVERT);
108    pio_gpio_init(pio, pin_scl);
109    gpio_set_oeover(pin_scl, GPIO_OVERRIDE_INVERT);
110    pio_sm_set_pins_with_mask(pio, sm, 0, both_pins);
111
112    // 启动前清除IRQ标志，并确保该标志不会实际触发
113    // 系统级中断（我们将其用作状态标志）
114    pio_set_irq0_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
115        sm), false);
115    pio_set_irq1_source_enabled(pio, (enum pio_interrupt_source) ((uint) pis_interrupt0 +
116        sm), false);
116    pio_interrupt_clear(pio, sm);
117
118    // 配置并启动SM
119    pio_sm_init(pio, sm, offset + i2c_offset_entry_point, &c);
120    pio_sm_set_enabled(pio, sm, true);
121 }

```

我们还可以使用 PIO 汇编器生成一张指令表，用于通过 FIFO 传递，适用于启动/停止/重新启动条件。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c.pio> 第126至136行

```

126 .program set_scl_sda
127 .side_set 1 opt
128
129 ; 汇编一张指令表，软件可从中选择并传入 FIFO，以发出 START/STOP/RSTART 指令。此表并非设计
130   为完整程序运行。
131
132
133     set pindirs, 0 side 0 [7] ; SCL = 0, SDA = 0
134     set pindirs, 1 side 0 [7] ; SCL = 0, SDA = 1
135     set pindirs, 0 side 1 [7] ; SCL = 1, SDA = 0
136     set pindirs, 1 side 1 [7] ; SCL = 1, SDA = 1

```

示例代码对状态机的FIFO执行阻塞式软件IO，以避免配置系统DMA的额外复杂性。例如，I2C启动条件的入队操作如下：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第69至73行

```

69 void pio_i2c_start(PIO pio, uint sm) {
70     pio_i2c_put_or_err(pio, sm, 1u << PIO_I2C_ICOUNT_LSB); // 2条指令的转义代码
71     // 序列
72     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC1_SD0]); // 我们已处于空闲状态
73     // , 仅需将SDA线拉低
74     pio_i2c_put_or_err(pio, sm, set_scl_sda_program_instructions[I2C_SC0_SD0]); // 同时拉低时钟线,
75     // 以便传输数据
76 }

```

由于I2C在多个环节可能出错，需能够检测状态机断言的错误标志，清除停止状态并重新启动，在断言停止条件及释放总线之前完成处理。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c Lines 15 - 17

```
15 bool pio_i2c_check_error(PIO pio, uint sm) {
16     return pio_interrupt_get(pio, sm);
17 }
```

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c Lines 19 - 23

```
19 void pio_i2c_resume_after_error(PIO pio, uint sm) {
20     pio_sm_drain_tx_fifo(pio, sm);
21     pio_sm_exec(pio, sm, (pio->sm[sm].execctrl & PIO_SM0_EXECCTRL_WRAP_BOTTOM_BITS) >>
22         PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
23     pio_interrupt_clear(pio, sm);
24 }
```

We need some higher-level functions to pass correctly-formatted data though the FIFOs and insert Starts, Stops, NAKs and so on at the correct points. This is enough to present a similar interface to the other hardware I2Cs on RP2040.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c_bus_scan.c Lines 13 - 42

```
13 int main() {
14     stdio_init_all();
15
16     PIO pio = pio0;
17     uint sm = 0;
18     uint offset = pio_add_program(pio, &i2c_program);
19     i2c_program_init(pio, sm, offset, PIN_SDA, PIN_SCL);
20
21     printf("\nPIO I2C Bus Scan\n");
22     printf("  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
23
24     for (int addr = 0; addr < (1 << 7); ++addr) {
25         if (addr % 16 == 0) {
26             printf("%02x ", addr);
27         }
28         // Perform a 0-byte read from the probe address. The read function
29         // returns a negative result NAK'd any time other than the last data
30         // byte. Skip over reserved addresses.
31         int result;
32         if (reserved_addr(addr))
33             result = -1;
34         else
35             result = pio_i2c_read_blocking(pio, sm, addr, NULL, 0);
36
37         printf(result < 0 ? "." : "@");
38         printf(addr % 16 == 15 ? "\n" : " ");
39     }
40     printf("Done.\n");
41     return 0;
42 }
```

3.6.8. PWM

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第15至17行

```
15 bool pio_i2c_check_error(PIO pio, uint sm) {
16     return pio_interrupt_get(pio, sm);
17 }
```

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/pio_i2c.c 第19至23行

```
19 void pio_i2c_resume_after_error(PIO pio, uint sm) {
20     pio_sm_drain_tx_fifo(pio, sm);
21     pio_sm_exec(pio, sm, (pio->sm[sm].execctrl & PIO_SM0_EXECCTRL_WRAP_BOTTOM_BITS) >>
22     PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
23     pio_interrupt_clear(pio, sm);
24 }
```

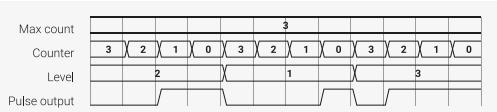
我们需要一些更高级的函数，以便通过FIFO传递格式正确的数据，并在正确位置插入起始位、停止位、NAK等信号。这足以提供与RP2040上其他硬件I2C类似的接口。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pio/i2c/i2c_bus_scan.c 第13至42行

```
13 int main() {
14     stdio_init_all();
15
16     PIO pio = pio0;
17     uint sm = 0;
18     uint offset = pio_add_program(pio, &i2c_program);
19     i2c_program_init(pio, sm, offset, PIN_SDA, PIN_SCL);
20
21     printf("\nPIO I2C总线扫描\n");
22     printf("  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
23
24     for (int addr = 0; addr < (1 << 7); ++addr) {
25         if (addr % 16 == 0) {
26             printf("%02x ", addr);
27         }
28         // 从探针地址执行0字节读取。读取函数
29         // 除最后一个数据字节外，任何时刻返回负结果即表示NAK。
30         // byte. 跳过保留地址。
31         int result;
32         if (reserved_addr(addr))
33             result = -1;
34         else
35             result = pio_i2c_read_blocking(pio, sm, addr, NULL, 0);
36
37         printf(result < 0 ? "." : "@");
38         printf(addr % 16 == 15 ? "\n" : " ");
39     }
40     printf("完成.\n");
41     return 0;
42 }
```

3.6.8. PWM

Figure 56. Pulse width modulation (PWM). The state machine outputs positive voltage pulses at regular intervals. The width of these pulses is controlled, so that the line is high for some controlled fraction of the time (the duty cycle). One use of this is to smoothly vary the brightness of an LED, by pulsing it faster than human persistence of vision.



This program repeatedly counts down to 0 with the Y register, whilst comparing the Y count to a pulse width held in the X register. The output is asserted low before counting begins, and asserted high when the value in Y reaches X. Once Y reaches 0, the process repeats, and the output is once more driven low. The fraction of time that the output is high is therefore proportional to the pulse width stored in X.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> Lines 10 - 22

```

10 .program pwm
11 .side_set 1 opt
12
13     pull noblock    side 0 ; Pull from FIFO to OSR if available, else copy X to OSR.
14     mov x, osr        ; Copy most-recently-pulled value back to scratch X
15     mov y, isr         ; ISR contains PWM period. Y used as counter.
16 countloop:
17     jmp x!=y noset    ; Set pin high if X == Y, keep the two paths length matched
18     jmp skip          side 1
19 noset:
20     nop               ; Single dummy cycle to keep the two paths the same length
21 skip:
22     jmp y-- countloop ; Loop until Y hits 0, then pull a fresh PWM value from FIFO

```

Often, a PWM can be left at a particular pulse width for thousands of pulses, rather than supplying a new pulse width each time. This example highlights how a nonblocking **PULL** (Section 3.4.7) can achieve this: if the TX FIFO is empty, a nonblocking **PULL** will copy X to the OSR. After pulling, the program copies the OSR into X, so that it can be compared to the count value in Y. The net effect is that, if a new duty cycle value has not been supplied through the TX FIFO at the start of this period, the duty cycle from the previous period (which has been copied from X to OSR via the failed **PULL**, and then back to X via the **MOV**) is reused, for as many periods as necessary.

Another useful technique shown here is using the ISR as a configuration register, if **IN** instructions are not required. System software can load an arbitrary 32-bit value into the ISR (by executing instructions directly on the state machine), and the program will copy this value into Y each time it begins counting. The ISR can be used to configure the range of PWM counting, and the state machine's clock divider controls the rate of counting.

To start modulating some pulses, we first need to map the state machine's side-set pins to the GPIO we want to output PWM on, and tell the state machine where the program is loaded in the PIO instruction memory:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> Lines 25 - 31

```

25 static inline void pwm_program_init(PIO pio, uint sm, uint offset, uint pin) {
26     pio_gpio_init(pio, pin);
27     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
28     pio_sm_config c = pwm_program_get_default_config(offset);
29     sm_config_set_sideset_pins(&c, pin);
30     pio_sm_init(pio, sm, offset, &c);
31 }

```

A little footwork is required to load the ISR with the desired counting range:

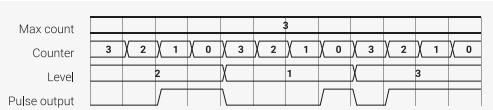
Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> Lines 14 - 20

```

14 void pio_pwm_set_period(PIO pio, uint sm, uint32_t period) {
15     pio_sm_set_enabled(pio, sm, false);
16     pio_sm_put_blocking(pio, sm, period);
17     pio_sm_exec(pio, sm, pio_encode_pull(false, false));

```

图56。脉宽调制(PWM)。状态机以固定间隔输出正电压脉冲。这些脉冲的宽度受控，使线路在一定时间内维持高电平(占空比)。其用途之一是通过脉冲频率快于人眼视觉暂留，从而平滑调节LED亮度。



该程序利用Y寄存器反复倒计时至0，并将Y计数与保存在X寄存器中的脉冲宽度进行比较。在计数开始前，输出被置为低电平；当Y的值达到X时，输出被置为高电平。一旦Y归零，该过程重复，输出再次置为低电平。因此，输出处于高电平的时间比例与存储在X中的脉冲宽度成正比。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> 第10至22行

```

10 .program pwm
11 .side_set 1 opt
12
13     pull noblock    side 0 ; 若 FIFO 中有数据，则从 FIFO 拉取至 OSR；否则将 X 复制至 OSR。
14     mov x, osr          ; 将最近拉取的值复制回暂存寄存器 X
15     mov y, isr          ; ISR 包含 PWM 周期，Y 用作计数器。
16 countloop:
17     jmp x!=y noset      ; 如果 X == Y，则将引脚设为高电平，保持两条路径长度匹配
18     jmp skip           side 1
19 noset:
20     nop                  ; 单个占位周期，用以保持两条路径长度一致
21 skip:
22     jmp y-- countloop    ; 循环直到 Y 达到 0，然后从 FIFO 拉取新的 PWM 值

```

通常，PWM 可在特定脉冲宽度下持续数千个脉冲，而无需每次供应新的脉冲宽度。本示例说明了非阻塞 **PULL** (第3.4.7节) 如何实现此功能：若 TX FIFO 为空，非阻塞 **PULL** 会将 X 复制至 OSR。拉取完成后，程序将 OSR 复制回 X，以便与 Y 中的计数值比较。其最终效果是：若本周期开始时 TX FIFO 未提供新的占空比值，则上一周期的占空比 (由失败的 **PULL** 从 X 复制至 OSR，再通过 **MOV** 复制回 X) 将被重复使用，持续所需周期数。

此处展示的另一项实用技术是将ISR用作配置寄存器，前提是无需使用 **IN** 指令。

系统软件可将任意32位数值加载至ISR (通过直接在状态机上执行指令)，并且程序将在每次开始计数时将该数值复制到 Y 寄存器。ISR可用于配置PWM计数范围，状态机的时钟分频器控制计数速率。

要开始调制脉冲，首先需将状态机的侧边引脚映射至目标输出PWM的GPIO，并告知状态机程序在PIO指令存储器中的加载地址：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.pio> 第25至31行

```

25 static inline void pwm_program_init(PIO pio, uint sm, uint offset, uint pin) {
26     pio_gpio_init(pio, pin);
27     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
28     pio_sm_config c = pwm_program_get_default_config(offset);
29     sm_config_set_sideset_pins(&c, pin);
30     pio_sm_init(pio, sm, offset, &c);
31 }

```

加载带有所需计数范围的ISR需要一些准备工作：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第14至20行

```

14 void pio_pwm_set_period(PIO pio, uint sm, uint32_t period) {
15     pio_sm_set_enabled(pio, sm, false);
16     pio_sm_put_blocking(pio, sm, period);
17     pio_sm_exec(pio, sm, pio_encode_pull(false, false));

```

```

18     pio_sm_exec(pio, sm, pio_encode_out(pio_isr, 32));
19     pio_sm_set_enabled(pio, sm, true);
20 }
```

Once this is done, the state machine can be enabled, and PWM values written directly to its TX FIFO.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> Lines 23 - 25

```

23 void pio_pwm_set_level(PIO pio, uint sm, uint32_t level) {
24     pio_sm_put_blocking(pio, sm, level);
25 }
```

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> Lines 27 - 51

```

27 int main() {
28     stdio_init_all();
29 #ifndef PICO_DEFAULT_LED_PIN
30 #warning pio/pwm example requires a board with a regular LED
31     puts("Default LED pin was not defined");
32 #else
33
34     // todo get free sm
35     PIO pio = pio0;
36     int sm = 0;
37     uint offset = pio_add_program(pio, &pwm_program);
38     printf("Loaded program at %d\n", offset);
39
40     pwm_program_init(pio, sm, offset, PICO_DEFAULT_LED_PIN);
41     pio_pwm_set_period(pio, sm, (1u << 16) - 1);
42
43     int level = 0;
44     while (true) {
45         printf("Level = %d\n", level);
46         pio_pwm_set_level(pio, sm, level * level);
47         level = (level + 1) % 256;
48         sleep_ms(10);
49     }
50 #endif
51 }
```

If the TX FIFO is kept topped up with fresh pulse width values, this program will consume a new pulse width for each pulse. Once the FIFO runs dry, the program will again start reusing the most recently supplied value.

3.6.9. Addition

Although not designed for computation, PIO is quite likely Turing-complete, provided a long enough piece of tape can be found. It is conjectured that it could run DOOM, given a sufficiently high clock speed.

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.pio> Lines 7 - 25

```

7 .program addition
8
9 ; Pop two 32 bit integers from the TX FIFO, add them together, and push the
10 ; result to the TX FIFO. Autopush/pull should be disabled as we're using
11 ; explicit push and pull instructions.
12 ;
```

```

18     pio_sm_exec(pio, sm, pio_encode_out(pio_isr, 32));
19     pio_sm_set_enabled(pio, sm, true);
20 }
```

完成上述操作后，状态机即可启用，并可直接向其TX FIFO写入PWM值。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第23至25行

```

23 void pio_pwm_set_level(PIO pio, uint sm, uint32_t level) {
24     pio_sm_put_blocking(pio, sm, level);
25 }
```

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/pwm/pwm.c> 第27行至51行

```

27 int main() {
28     stdio_init_all();
29 #ifndef PICO_DEFAULT_LED_PIN
30 #warning pio/pwm示例需配备带有普通LED的开发板
31     puts("默认LED引脚未定义");
32 #else
33
34 // 待办：获取可用的sm
35 PIO pio = pio0;
36 int sm = 0;
37 uint offset = pio_add_program(pio, &pwm_program);
38 printf("已加载程序，偏移量: %d\n", offset);
39
40 pwm_program_init(pio, sm, offset, PICO_DEFAULT_LED_PIN);
41 pio_pwm_set_period(pio, sm, (1u << 16) - 1);
42
43 int level = 0;
44 while (true) {
45     printf("Level = %d\n", level);
46     pio_pwm_set_level(pio, sm, level * level);
47     level = (level + 1) % 256;
48     sleep_ms(10);
49 }
50#endif
51 }
```

如果 TX FIFO 始终保持填充最新的脉宽值，则该程序将在每个脉冲周期消耗一个新的脉宽值。一旦 FIFO 为空，程序将重新使用最近提供的值。

3.6.9. 加法

尽管 PIO 并非专为计算设计，但只要存在足够长的带子，其极有可能具备图灵完备性。有人推测，在足够高的时钟频率下，它甚至可能运行 DOOM。

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.pio> 第7至25行

```

7 .program addition
8
9 ; 从 TX FIFO 弹出两个 32 位整数，将其相加，并将结果推入 TX FIFO。由于正在使用自动推送/拉取
功能，应将其禁用。
11 ; 显示的推拉指令。
12 ;
```

```

13 ; This program uses the two's complement identity x + y == ~(~x - y)
14
15     pull
16     mov x, ~osr
17     pull
18     mov y, osr
19     jmp test      ; this loop is equivalent to the following C code:
20 incr:           ; while (y--)
21     jmp x-- test ;     x--;
22 test:          ; This has the effect of subtracting y from x, eventually.
23     jmp y-- incr
24     mov isr, ~x
25     push

```

A full 32-bit addition takes only around one minute at 125MHz. The program pulls two numbers from the TX FIFO and pushes their sum to the RX FIFO, which is perfect for use either with the system DMA, or directly by the processor:

Pico Examples: <https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.c>

```

1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "hardware/pio.h"
12 #include "addition.pio.h"
13
14 // Pop quiz: how many additions does the processor do when calling this function
15 uint32_t do_addition(PIO pio, uint sm, uint32_t a, uint32_t b) {
16     pio_sm_put_blocking(pio, sm, a);
17     pio_sm_put_blocking(pio, sm, b);
18     return pio_sm_get_blocking(pio, sm);
19 }
20
21 int main() {
22     stdio_init_all();
23
24     PIO pio = pio0;
25     uint sm = 0;
26     uint offset = pio_add_program(pio, &addition_program);
27     addition_program_init(pio, sm, offset);
28
29     printf("Doing some random additions:\n");
30     for (int i = 0; i < 10; ++i) {
31         uint a = rand() % 100;
32         uint b = rand() % 100;
33         printf("%u + %u = %u\n", a, b, do_addition(pio, sm, a, b));
34     }
35 }

```

3.6.10. Further Examples

The **Raspberry Pi Pico-series C/C++ SDK** book has a PIO chapter which goes into depth on some software-centric topics not presented here. It includes a PIO + DMA logic analyser example that can sample every GPIO on every cycle (a

```

13 ; 该程序使用二补码恒等式 x + y == ~(~x - y)
14
15 拉出
16 mov x, ~osr
17 拉出
18 mov y, osr
19 jmp test      ; 该循环等价于以下C语言代码:
20 incr:          ; while (y--)
21 jmp x-- test   ;     x--;
22 test:          ; 此操作最终实现从 x 中减去 y 的效果。
23 jmp y-- incr
24 mov isr, ~x
25 推入

```

在125MHz频率下，完整的32位加法仅需约一分钟。程序从TX FIFO拉取两个数字，并将其和推入RX FIFO，非常适合与系统DMA或处理器直接配合使用：

Pico示例：<https://github.com/raspberrypi/pico-examples/blob/master/pio/addition/addition.c>

```

1 /**
2 * 版权所有 (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX许可证标识符: BSD-3-Clause
5 */
6
7 #include <stdlib.h>
8 #include <stdio.h>
9
10 #include "pico/stdlib.h"
11 #include "hardware/pio.h"
12 #include "addition.pio.h"
13
14 // 快答测验：调用此函数时处理器执行了多少次加法？
15 uint32_t do_addition(PIO pio, uint sm, uint32_t a, uint32_t b) {
16     pio_sm_put_blocking(pio, sm, a);
17     pio_sm_put_blocking(pio, sm, b);
18     return pio_sm_get_blocking(pio, sm);
19 }
20
21 int main() {
22     stdio_init_all();
23
24     PIO pio = pio0;
25     uint sm = 0;
26     uint offset = pio_add_program(pio, &addition_program);
27     addition_program_init(pio, sm, offset);
28
29     printf("进行一些随机加法运算: \n");
30     for (int i = 0; i < 10; ++i) {
31         uint a = rand() % 100;
32         uint b = rand() % 100;
33         printf("%u + %u = %u\n", a, b, do_addition(pio, sm, a, b));
34     }
35 }

```

3.6.10. 更多示例

树莓派 Pico 系列 **C/C++ SDK** 手册中设有一章详尽介绍 **PIO**，涵盖本处未呈现的部分软件核心内容。其中包含一个 **PIO + DMA** 逻辑分析仪示例，可在每个周期采样所有 **GPIO**（

bandwidth of nearly 4Gbps at 125MHz, although this does fill up RP2040's RAM quite quickly).

There are also further examples in the [pio/](#) directory in the [Pico Examples](#) repository.

Some of the more experimental example code, such as DPI and SD card support, is currently located in the [Pico Extras](#) and [Pico Playground](#) repositories. The PIO parts of these are functional, but the surrounding software stacks are still in an experimental state.

3.7. List of Registers

The PIO0 and PIO1 registers start at base addresses of `0x50200000` and `0x50300000` respectively (defined as `PIO0_BASE` and `PIO1_BASE` in SDK).

Table 367. List of PIO registers

Offset	Name	Info
0x000	CTRL	PIO control register
0x004	FSTAT	FIFO status register
0x008	FDEBUG	FIFO debug register
0x00c	FLEVEL	FIFO levels
0x010	TXF0	Direct write access to the TX FIFO for this state machine. Each write pushes one word to the FIFO. Attempting to write to a full FIFO has no effect on the FIFO state or contents, and sets the sticky FDEBUG_TXOVER error flag for this FIFO.
0x014	TXF1	Direct write access to the TX FIFO for this state machine. Each write pushes one word to the FIFO. Attempting to write to a full FIFO has no effect on the FIFO state or contents, and sets the sticky FDEBUG_TXOVER error flag for this FIFO.
0x018	TXF2	Direct write access to the TX FIFO for this state machine. Each write pushes one word to the FIFO. Attempting to write to a full FIFO has no effect on the FIFO state or contents, and sets the sticky FDEBUG_TXOVER error flag for this FIFO.
0x01c	TXF3	Direct write access to the TX FIFO for this state machine. Each write pushes one word to the FIFO. Attempting to write to a full FIFO has no effect on the FIFO state or contents, and sets the sticky FDEBUG_TXOVER error flag for this FIFO.
0x020	RXF0	Direct read access to the RX FIFO for this state machine. Each read pops one word from the FIFO. Attempting to read from an empty FIFO has no effect on the FIFO state, and sets the sticky FDEBUG_RXUNDER error flag for this FIFO. The data returned to the system on a read from an empty FIFO is undefined.
0x024	RXF1	Direct read access to the RX FIFO for this state machine. Each read pops one word from the FIFO. Attempting to read from an empty FIFO has no effect on the FIFO state, and sets the sticky FDEBUG_RXUNDER error flag for this FIFO. The data returned to the system on a read from an empty FIFO is undefined.
0x028	RXF2	Direct read access to the RX FIFO for this state machine. Each read pops one word from the FIFO. Attempting to read from an empty FIFO has no effect on the FIFO state, and sets the sticky FDEBUG_RXUNDER error flag for this FIFO. The data returned to the system on a read from an empty FIFO is undefined.

带宽近 4Gbps 于 125MHz，尽管这会迅速占用 RP2040 的 RAM)。

Pico 示例仓库的 [pio/](#) 目录中亦有更多示例。

部分较为实验性的示例代码，如 DPI 和 SD 卡支持，现存于 Pico Extras 与 Pico Playground 仓库中。这些 PIO 部分具有功能性，但其周边软件堆栈仍处于实验阶段。

3.7. 寄存器列表

PIO0 和 PIO1 寄存器的地址分别为 `0x50200000` 和 `0x50300000` (在 SDK 中定义为 `PIO0_BASE` 和 `PIO1_BASE`)。

表 367: PIO 寄存器列表

偏移量	名称	说明
0x000	CTRL	PIO 控制寄存器
0x004	FSTAT	FIFO 状态寄存器
0x008	FDEBUG	FIFO 调试寄存器
0x00c	FLEVEL	FIFO 水位
0x010	TXF0	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x014	TXF1	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x018	TXF2	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x01c	TXF3	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。
0x020	RXF0	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x024	RXF1	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x028	RXF2	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。

Offset	Name	Info
0x02c	RXF3	Direct read access to the RX FIFO for this state machine. Each read pops one word from the FIFO. Attempting to read from an empty FIFO has no effect on the FIFO state, and sets the sticky FDEBUG_RXUNDER error flag for this FIFO. The data returned to the system on a read from an empty FIFO is undefined.
0x030	IRQ	<p>State machine IRQ flags register. Write 1 to clear. There are 8 state machine IRQ flags, which can be set, cleared, and waited on by the state machines. There's no fixed association between flags and state machines – any state machine can use any flag.</p> <p>Any of the 8 flags can be used for timing synchronisation between state machines, using IRQ and WAIT instructions. The lower four of these flags are also routed out to system-level interrupt requests, alongside FIFO status interrupts – see e.g. IRQ0_INTE.</p>
0x034	IRQ_FORCE	Writing a 1 to each of these bits will forcibly assert the corresponding IRQ. Note this is different to the INTF register: writing here affects PIO internal state. INTF just asserts the processor-facing IRQ signal for testing ISRs, and is not visible to the state machines.
0x038	INPUT_SYNC_BYPASS	<p>There is a 2-flipflop synchronizer on each GPIO input, which protects PIO logic from metastabilities. This increases input delay, and for fast synchronous IO (e.g. SPI) these synchronizers may need to be bypassed. Each bit in this register corresponds to one GPIO.</p> <p>0 → input is synchronized (default) 1 → synchronizer is bypassed If in doubt, leave this register as all zeroes.</p>
0x03c	DBG_PADOUT	Read to sample the pad output values PIO is currently driving to the GPIOs. On RP2040 there are 30 GPIOs, so the two most significant bits are hardwired to 0.
0x040	DBG_PADOE	Read to sample the pad output enables (direction) PIO is currently driving to the GPIOs. On RP2040 there are 30 GPIOs, so the two most significant bits are hardwired to 0.
0x044	DBG_CFGINFO	<p>The PIO hardware has some free parameters that may vary between chip products.</p> <p>These should be provided in the chip datasheet, but are also exposed here.</p>
0x048	INSTR_MEM0	Write-only access to instruction memory location 0
0x04c	INSTR_MEM1	Write-only access to instruction memory location 1
0x050	INSTR_MEM2	Write-only access to instruction memory location 2
0x054	INSTR_MEM3	Write-only access to instruction memory location 3
0x058	INSTR_MEM4	Write-only access to instruction memory location 4
0x05c	INSTR_MEM5	Write-only access to instruction memory location 5
0x060	INSTR_MEM6	Write-only access to instruction memory location 6
0x064	INSTR_MEM7	Write-only access to instruction memory location 7

偏移量	名称	说明
0x02c	RXF3	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBUG_RXUNDER 错误标志。从空 FIFO 读取返回给系统的数据未定义。
0x030	IRQ	状态机中断请求标志寄存器。写入 1 可清除。共有 8 个状态机中断请求标志，状态机可对其进行设置、清除和等待。标志与状态机之间无固定对应关系——任意状态机均可使用任何标志。 这 8 个标志中的任意一个均可用于状态机间的时间同步，配合中断（IRQ）和等待（WAIT）指令使用。其中较低的四个标志还被路由至系统级中断请求，与 FIFO 状态中断一并触发——详见如 IRQ0_INTE。
0x034	IRQ_FORCE	向这些位写入 1 会强制触发对应的中断请求（IRQ）。请注意，此操作与 INTF 寄存器不同：此处写入会影响 PIO 的内部状态。INTF 仅断言面向处理器的 IRQ 信号以测试 ISR，状态机不可见该信号。
0x038	INPUT_SYNC_BYPASS	每个 GPIO 输入端均设有 2 级触发器同步器，以防止 PIO 逻辑发生亚稳态。这会增加输入延迟，对于高速同步 IO（如 SPI），可能需要绕过该同步器。该寄存器中的每个位对应一个 GPIO。 0 → 输入已同步（默认） 1 → 同步器已被绕过 如有疑问，请保持此寄存器全零。
0x03c	DBG_PADOUT	读取以采样 PIO 当前驱动至 GPIO 的引脚输出值。RP2040 共有 30 个 GPIO，最高两位硬连线为 0。
0x040	DBG_PADOE	读取以采样 PIO 当前驱动至 GPIO 的引脚输出使能（方向）。RP2040 共有 30 个 GPIO，最高两位硬连线为 0。
0x044	DBG_CFGINFO	PIO 硬件具有一些在不同芯片产品之间可能变化的自由参数。 这些参数应在芯片数据手册中提供，但此处亦有展示。
0x048	INSTR_MEM0	指令存储位置 0 的写入专用访问
0x04c	INSTR_MEM1	指令存储位置 1 的写入专用访问
0x050	INSTR_MEM2	指令存储位置 2 的写入专用访问
0x054	INSTR_MEM3	指令存储位置 3 的写入专用访问
0x058	INSTR_MEM4	指令存储位置 4 的写入专用访问
0x05c	INSTR_MEM5	指令存储位置 5 的写入专用访问
0x060	INSTR_MEM6	指令存储位置 6 的写入专用访问
0x064	INSTR_MEM7	指令存储位置 7 的写入专用访问

Offset	Name	Info
0x068	INSTR_MEM8	Write-only access to instruction memory location 8
0x06c	INSTR_MEM9	Write-only access to instruction memory location 9
0x070	INSTR_MEM10	Write-only access to instruction memory location 10
0x074	INSTR_MEM11	Write-only access to instruction memory location 11
0x078	INSTR_MEM12	Write-only access to instruction memory location 12
0x07c	INSTR_MEM13	Write-only access to instruction memory location 13
0x080	INSTR_MEM14	Write-only access to instruction memory location 14
0x084	INSTR_MEM15	Write-only access to instruction memory location 15
0x088	INSTR_MEM16	Write-only access to instruction memory location 16
0x08c	INSTR_MEM17	Write-only access to instruction memory location 17
0x090	INSTR_MEM18	Write-only access to instruction memory location 18
0x094	INSTR_MEM19	Write-only access to instruction memory location 19
0x098	INSTR_MEM20	Write-only access to instruction memory location 20
0x09c	INSTR_MEM21	Write-only access to instruction memory location 21
0x0a0	INSTR_MEM22	Write-only access to instruction memory location 22
0x0a4	INSTR_MEM23	Write-only access to instruction memory location 23
0x0a8	INSTR_MEM24	Write-only access to instruction memory location 24
0x0ac	INSTR_MEM25	Write-only access to instruction memory location 25
0x0b0	INSTR_MEM26	Write-only access to instruction memory location 26
0x0b4	INSTR_MEM27	Write-only access to instruction memory location 27
0x0b8	INSTR_MEM28	Write-only access to instruction memory location 28
0x0bc	INSTR_MEM29	Write-only access to instruction memory location 29
0x0c0	INSTR_MEM30	Write-only access to instruction memory location 30
0x0c4	INSTR_MEM31	Write-only access to instruction memory location 31
0x0c8	SM0_CLKDIV	Clock divisor register for state machine 0 Frequency = clock freq / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0cc	SM0_EXECCTRL	Execution/behavioural settings for state machine 0
0x0d0	SM0_SHIFTCTRL	Control behaviour of the input/output shift registers for state machine 0
0x0d4	SM0_ADDR	Current instruction address of state machine 0
0x0d8	SM0_INSTR	Read to see the instruction currently addressed by state machine 0's program counter Write to execute an instruction immediately (including jumps) and then resume execution.
0x0dc	SM0_PINCTRL	State machine pin control
0x0e0	SM1_CLKDIV	Clock divisor register for state machine 1 Frequency = clock freq / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0e4	SM1_EXECCTRL	Execution/behavioural settings for state machine 1

偏移量	名称	说明
0x068	INSTR_MEM8	指令存储位置 8 的写入专用访问
0x06c	INSTR_MEM9	指令存储位置 9 的写入专用访问
0x070	INSTR_MEM10	指令存储位置 10 的写入专用访问
0x074	INSTR_MEM11	指令存储位置 11 的写入专用访问
0x078	INSTR_MEM12	指令存储位置 12 的写入专用访问
0x07c	INSTR_MEM13	指令存储位置 13 的写入专用访问
0x080	INSTR_MEM14	指令存储位置 14 的写入专用访问
0x084	INSTR_MEM15	指令存储位置 15 的写入专用访问
0x088	INSTR_MEM16	对指令内存地址16的仅写访问
0x08c	INSTR_MEM17	对指令内存地址17的仅写访问
0x090	INSTR_MEM18	对指令内存地址18的仅写访问
0x094	INSTR_MEM19	对指令内存地址19的仅写访问
0x098	INSTR_MEM20	对指令内存地址20的仅写访问
0x09c	INSTR_MEM21	对指令内存地址21的仅写访问
0x0a0	INSTR_MEM22	对指令内存地址22的仅写访问
0x0a4	INSTR_MEM23	对指令内存地址23的仅写访问
0x0a8	INSTR_MEM24	对指令内存地址24的仅写访问
0x0ac	INSTR_MEM25	对指令内存地址25的仅写访问
0x0b0	INSTR_MEM26	对指令内存地址26的仅写访问
0x0b4	INSTR_MEM27	对指令内存地址27的仅写访问
0x0b8	INSTR_MEM28	对指令内存地址28的仅写访问
0x0bc	INSTR_MEM29	对指令内存地址29的仅写访问
0x0c0	INSTR_MEM30	对指令内存地址30的仅写访问
0x0c4	INSTR_MEM31	对指令内存地址31的仅写访问
0x0c8	SM0_CLKDIV	状态机0的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0cc	SM0_EXECCTRL	状态机0的执行/行为设置
0x0d0	SM0_SHIFTCTRL	控制状态机0输入/输出移位寄存器的行为
0x0d4	SM0_ADDR	状态机0的当前指令地址
0x0d8	SM0_INSTR	读取以查看状态机当前指向的指令 0号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x0dc	SM0_PINCTRL	状态机引脚控制
0x0e0	SM1_CLKDIV	状态机1的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0e4	SM1_EXECCTRL	状态机1的执行/行为设置

Offset	Name	Info
0x0e8	SM1_SHIFTCTRL	Control behaviour of the input/output shift registers for state machine 1
0x0ec	SM1_ADDR	Current instruction address of state machine 1
0x0f0	SM1_INSTR	Read to see the instruction currently addressed by state machine 1's program counter Write to execute an instruction immediately (including jumps) and then resume execution.
0x0f4	SM1_PINCTRL	State machine pin control
0x0f8	SM2_CLKDIV	Clock divisor register for state machine 2 Frequency = clock freq / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0fc	SM2_EXECCTRL	Execution/behavioural settings for state machine 2
0x100	SM2_SHIFTCTRL	Control behaviour of the input/output shift registers for state machine 2
0x104	SM2_ADDR	Current instruction address of state machine 2
0x108	SM2_INSTR	Read to see the instruction currently addressed by state machine 2's program counter Write to execute an instruction immediately (including jumps) and then resume execution.
0x10c	SM2_PINCTRL	State machine pin control
0x110	SM3_CLKDIV	Clock divisor register for state machine 3 Frequency = clock freq / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x114	SM3_EXECCTRL	Execution/behavioural settings for state machine 3
0x118	SM3_SHIFTCTRL	Control behaviour of the input/output shift registers for state machine 3
0x11c	SM3_ADDR	Current instruction address of state machine 3
0x120	SM3_INSTR	Read to see the instruction currently addressed by state machine 3's program counter Write to execute an instruction immediately (including jumps) and then resume execution.
0x124	SM3_PINCTRL	State machine pin control
0x128	INTR	Raw Interrupts
0x12c	IRQ0_INTE	Interrupt Enable for irq0
0x130	IRQ0_INTF	Interrupt Force for irq0
0x134	IRQ0_INTS	Interrupt status after masking & forcing for irq0
0x138	IRQ1_INTE	Interrupt Enable for irq1
0x13c	IRQ1_INTF	Interrupt Force for irq1
0x140	IRQ1_INTS	Interrupt status after masking & forcing for irq1

PIO: CTRL Register

Offset: 0x000

偏移量	名称	说明
0x0e8	SM1_SHIFTCTRL	控制状态机1输入/输出移位寄存器的行为
0x0ec	SM1_ADDR	状态机1的当前指令地址
0x0f0	SM1_INSTR	读取以查看状态机当前指向的指令 1号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x0f4	SM1_PINCTRL	状态机引脚控制
0x0f8	SM2_CLKDIV	状态机2的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x0fc	SM2_EXECCTRL	状态机2的执行/行为设置
0x100	SM2_SHIFTCTRL	控制状态机2输入/输出移位寄存器的行为
0x104	SM2_ADDR	状态机2的当前指令地址
0x108	SM2_INSTR	读取以查看状态机当前指向的指令 2号程序计数器 写入以立即执行指令（包括跳转）并随后恢复执行。
0x10c	SM2_PINCTRL	状态机引脚控制
0x110	SM3_CLKDIV	状态机3的时钟分频寄存器 频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)
0x114	SM3_EXECCTRL	状态机3的执行/行为设置
0x118	SM3_SHIFTCTRL	控制状态机3输入/输出移位寄存器的行为
0x11c	SM3_ADDR	状态机3的当前指令地址
0x120	SM3_INSTR	读取状态机3程序计数器当前指向的指令 写入以立即执行指令（包括跳转）并随后恢复执行。
0x124	SM3_PINCTRL	状态机引脚控制
0x128	INTR	原始中断
0x12c	IRQ0_INTE	irq0中断使能
0x130	IRQ0_INTF	irq0中断强制
0x134	IRQ0_INTS	irq0中断屏蔽及强制后的状态
0x138	IRQ1_INTE	irq1中断使能
0x13c	IRQ1_INTF	irq1中断强制
0x140	IRQ1_INTS	irq1中断屏蔽及强制后的状态

PIO: CTRL寄存器

偏移: 0x000

Description

PIO control register

Table 368. CTRL Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:8	<p>CLKDIV_RESTART: Restart a state machine's clock divider from an initial phase of 0. Clock dividers are free-running, so once started, their output (including fractional jitter) is completely determined by the integer/fractional divisor configured in SMx_CLKDIV. This means that, if multiple clock dividers with the same divisor are restarted simultaneously, by writing multiple 1 bits to this field, the execution clocks of those state machines will run in precise lockstep.</p> <p>Note that setting/clearing SM_ENABLE does not stop the clock divider from running, so once multiple state machines' clocks are synchronised, it is safe to disable/reenable a state machine, whilst keeping the clock dividers in sync.</p> <p>Note also that CLKDIV_RESTART can be written to whilst the state machine is running, and this is useful to resynchronise clock dividers after the divisors (SMx_CLKDIV) have been changed on-the-fly.</p>	SC	0x0
7:4	<p>SM_RESTART: Write 1 to instantly clear internal SM state which may be otherwise difficult to access and will affect future execution.</p> <p>Specifically, the following are cleared: input and output shift counters; the contents of the input shift register; the delay counter; the waiting-on-IRQ state; any stalled instruction written to SMx_INSTR or run by OUT/MOV EXEC; any pin write left asserted due to OUT_STICKY.</p> <p>The program counter, the contents of the output shift register and the X/Y scratch registers are not affected.</p>	SC	0x0
3:0	<p>SM_ENABLE: Enable/disable each of the four state machines by writing 1/0 to each of these four bits. When disabled, a state machine will cease executing instructions, except those written directly to SMx_INSTR by the system.</p> <p>Multiple bits can be set/cleared at once to run/halt multiple state machines simultaneously.</p>	RW	0x0

PIO: FSTAT Register

Offset: 0x004

Description

FIFO status register

Table 369. FSTAT Register

Bits	Description	Type	Reset
31:28	Reserved.	-	-
27:24	TXEMPTY: State machine TX FIFO is empty	RO	0xf
23:20	Reserved.	-	-
19:16	TXFULL: State machine TX FIFO is full	RO	0x0
15:12	Reserved.	-	-
11:8	RXEMPTY: State machine RX FIFO is empty	RO	0xf
7:4	Reserved.	-	-

描述

PIO控制寄存器

表368. CTRL
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:8	<p>CLKDIV_RESTART: 重新启动状态机的时钟分频器，起始相位为0。时钟分频器为自由运行模式，启动后，其输出（包括分数抖动）完全由SMx_CLKDIV中配置的整数/分数分频系数决定。这意味着，若同时通过向此字段写入多个“1”位重新启动多个具有相同分频系数的时钟分频器，则这些状态机的执行时钟将实现精确同步。</p> <p>请注意，设置或清除SM_ENABLE并不会停止时钟分频器的运行，因此一旦多个状态机的时钟同步，禁用或重新启用状态机是安全的，同时保持时钟分频器同步。</p> <p>另请注意，可在状态机运行时写入CLKDIV_RESTART，此操作有助于在动态更改除数(SMx_CLKDIV)后重新同步时钟分频器。</p>	SC	0x0
7:4	<p>SM_RESTART: 写入1可立即清除内部状态机状态，该状态通常难以访问且会影响后续执行。</p> <p>具体而言，以下内容将被清除：输入和输出移位计数器；输入移位寄存器内容；延迟计数器；等待IRQ状态；任何写入至SMx_INSTR或由OUT/MOV EXEC执行的停滞指令；因OUT_STICKY导致仍保持断言状态的引脚写入。</p> <p>程序计数器、输出移位寄存器内容及X/Y临时寄存器不受影响。</p>	SC	0x0
3:0	<p>SM_ENABLE: 通过向这四个位分别写入1或0，启用或禁用各状态机。禁用状态机后，除系统直接写入SMx_INSTR的指令外，该状态机将停止执行指令。</p> <p>可同时设置或清除多个位，以同步启动或停止多个状态机。</p>	读写	0x0

PIO: FSTAT寄存器

偏移量: 0x004

描述

FIFO状态寄存器

表369: FSTAT
寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	TXEMPTY : 状态机的TX FIFO为空	只读	0xf
23:20	保留。	-	-
19:16	TXFULL : 状态机的TX FIFO已满	只读	0x0
15:12	保留。	-	-
11:8	RXEMPTY : 状态机的RX FIFO为空	只读	0xf
7:4	保留。	-	-

Bits	Description	Type	Reset
3:0	RXFULL: State machine RX FIFO is full	RO	0x0

PIO: FDEBUG Register

Offset: 0x008

Description

FIFO debug register

Table 370. FDEBUG Register

Bits	Description	Type	Reset
31:28	Reserved.	-	-
27:24	TXSTALL: State machine has stalled on empty TX FIFO during a blocking PULL, or an OUT with autopull enabled. Write 1 to clear.	WC	0x0
23:20	Reserved.	-	-
19:16	TXOVER: TX FIFO overflow (i.e. write-on-full by the system) has occurred. Write 1 to clear. Note that write-on-full does not alter the state or contents of the FIFO in any way, but the data that the system attempted to write is dropped, so if this flag is set, your software has quite likely dropped some data on the floor.	WC	0x0
15:12	Reserved.	-	-
11:8	RXUNDER: RX FIFO underflow (i.e. read-on-empty by the system) has occurred. Write 1 to clear. Note that read-on-empty does not perturb the state of the FIFO in any way, but the data returned by reading from an empty FIFO is undefined, so this flag generally only becomes set due to some kind of software error.	WC	0x0
7:4	Reserved.	-	-
3:0	RXSTALL: State machine has stalled on full RX FIFO during a blocking PUSH, or an IN with autopush enabled. This flag is also set when a nonblocking PUSH to a full FIFO took place, in which case the state machine has dropped data. Write 1 to clear.	WC	0x0

PIO: FLEVEL Register

Offset: 0x00c

Description

FIFO levels

Table 371. FLEVEL Register

Bits	Description	Type	Reset
31:28	RX3	RO	0x0
27:24	TX3	RO	0x0
23:20	RX2	RO	0x0
19:16	TX2	RO	0x0
15:12	RX1	RO	0x0
11:8	TX1	RO	0x0
7:4	RX0	RO	0x0
3:0	TX0	RO	0x0

位	描述	类型	复位值
3:0	RXFULL : 状态机的RX FIFO已满	只读	0x0

PIO: FDEBUG 寄存器

偏移量: 0x008

描述

FIFO调试寄存器

表 370: FDEBUG 寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	TXSTALL : 状态机在阻塞型PULL操作或启用自动拉取的OUT操作时, 因TX FIFO为空而暂停。写入1以清除该状态。	WC	0x0
23:20	保留。	-	-
19:16	TXOVER : 发生了TX FIFO溢出(即系统在FIFO满时尝试写入)。 写入1以清除该标志。请注意, 写入满时操作不会以任何方式改变FIFO的状态或内容, 但系统尝试写入的数据将被丢弃, 因此若该标志被置位, 您的软件很可能已丢失部分数据。	WC	0x0
15:12	保留。	-	-
11:8	RXUNDER : 发生了RX FIFO欠载(即系统在FIFO为空时尝试读取)。 写入1以清除该标志。请注意, 读取空时操作不会影响FIFO的状态, 但从空FIFO读取的数据是未定义的, 因此该标志通常仅因软件故障而置位。	WC	0x0
7:4	保留。	-	-
3:0	RXSTALL : 状态机在阻塞PUSH操作过程中, 或启用自动推送的IN操作时, 因RX FIFO已满而暂停。当对满FIFO执行非阻塞PUSH操作时, 该标志亦会置位, 此时状态机已丢弃数据。写入1以清除该标志。	WC	0x0

PIO: FLEVEL 寄存器

偏移: 0x00c

描述

FIFO水位

表 371: FLEVEL 寄存器

位	描述	类型	复位值
31:28	RX3	只读	0x0
27:24	TX3	只读	0x0
23:20	RX2	只读	0x0
19:16	TX2	只读	0x0
15:12	RX1	只读	0x0
11:8	TX1	只读	0x0
7:4	RX0	只读	0x0
3:0	TX0	只读	0x0

PIO: TXF0, TXF1, TXF2, TXF3 Registers

Offsets: 0x010, 0x014, 0x018, 0x01c

Table 372. TXF0, TXF1, TXF2, TXF3 Registers

Bits	Description	Type	Reset
31:0	Direct write access to the TX FIFO for this state machine. Each write pushes one word to the FIFO. Attempting to write to a full FIFO has no effect on the FIFO state or contents, and sets the sticky FDEBUG_TXOVER error flag for this FIFO.	WF	0x00000000

PIO: RXF0, RXF1, RXF2, RXF3 Registers

Offsets: 0x020, 0x024, 0x028, 0x02c

Table 373. RXF0, RXF1, RXF2, RXF3 Registers

Bits	Description	Type	Reset
31:0	Direct read access to the RX FIFO for this state machine. Each read pops one word from the FIFO. Attempting to read from an empty FIFO has no effect on the FIFO state, and sets the sticky FDEBUG_RXUNDER error flag for this FIFO. The data returned to the system on a read from an empty FIFO is undefined.	RF	-

PIO: IRQ Register

Offset: 0x030

Table 374. IRQ Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	<p>State machine IRQ flags register. Write 1 to clear. There are 8 state machine IRQ flags, which can be set, cleared, and waited on by the state machines. There's no fixed association between flags and state machines – any state machine can use any flag.</p> <p>Any of the 8 flags can be used for timing synchronisation between state machines, using IRQ and WAIT instructions. The lower four of these flags are also routed out to system-level interrupt requests, alongside FIFO status interrupts – see e.g. IRQ0_INTE.</p>	WC	0x00

PIO: IRQ_FORCE Register

Offset: 0x034

Table 375. IRQ_FORCE Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	<p>Writing a 1 to each of these bits will forcibly assert the corresponding IRQ. Note this is different to the INTF register: writing here affects PIO internal state. INTF just asserts the processor-facing IRQ signal for testing ISRs, and is not visible to the state machines.</p>	WF	0x00

PIO: INPUT_SYNC_BYPASS Register

Offset: 0x038

Table 376. INPUT_SYNC_BYPASS Register

PIO: TXF0、TXF1、TXF2、TXF3寄存器

偏移: 0x010、0x014、0x018、0x01c

表372. TXF0、TXF1、TXF2、TXF3寄存器

位	描述	类型	复位值
31:0	对该状态机的 TX FIFO 进行直接写访问。每次写操作均向 FIFO 推入一个字。尝试写入已满的 FIFO 不会改变 FIFO 状态或内容，并会为该 FIFO 设置粘性 FDEBUG_TXOVER 错误标志。	WF	0x00000000

PIO: RXF0、RXF1、RXF2、RXF3寄存器

偏移: 0x020、0x024、0x028、0x02c

表373. RXF0、RXF1、RXF2、RXF3寄存器

位	描述	类型	复位值
31:0	对该状态机的 RX FIFO 进行直接读访问。每次读取都会从 FIFO 中弹出一个字。尝试从空的 FIFO 读取不会改变 FIFO 状态，并会为该 FIFO 设置粘性 FDEBU G_RXUNDER 错误标志。 从空的 FIFO 读取返回给系统的数据为未定义。	RF	-

PIO: IRQ 寄存器

偏移: 0x030

表 374. IRQ 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	状态机中断请求标志寄存器。写入 1 可清除。共有 8 个状态机 IRQ 标志，可由状态机设置、清除及等待。 标志与状态机之间无固定对应关系——任意状态机均可使用任何标志。 这 8 个标志中的任意一个均可用于状态机间的时间同步，配合中断（IRQ）和等待（WAIT）指令使用。其中较低的四个标志还被路由至系统级中断请求，与 FIFO 状态中断一并触发——详见如 IRQ0_INTE。	WC	0x00

PIO: IRQ_FORCE 寄存器

偏移: 0x034

表 375. IRQ_FORCE 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	向这些位写入 1 将强制断言相应的 IRQ。 注意，此寄存器与 INTF 寄存器不同：在此写入会影响 PIO 内部状态。INTF 仅断言面向处理器的 IRQ 信号以测试 ISR，状态机不可见该信号。	WF	0x00

PIO: INPUT_SYNC_BYPASS 寄存器

偏移量: 0x038

表 376.
INPUT_SYNC_BYPASS
寄存器

Bits	Description	Type	Reset
31:0	<p>There is a 2-flipflop synchronizer on each GPIO input, which protects PIO logic from metastabilities. This increases input delay, and for fast synchronous IO (e.g. SPI) these synchronizers may need to be bypassed. Each bit in this register corresponds to one GPIO.</p> <p>0 → input is synchronized (default) 1 → synchronizer is bypassed If in doubt, leave this register as all zeroes.</p>	RW	0x00000000

PIO: DBG_PADOUT Register

Offset: 0x03c

Table 377.
DBG_PADOUT Register

Bits	Description	Type	Reset
31:0	Read to sample the pad output values PIO is currently driving to the GPIOs. On RP2040 there are 30 GPIOs, so the two most significant bits are hardwired to 0.	RO	0x00000000

PIO: DBG_PADOE Register

Offset: 0x040

Table 378.
DBG_PADOE Register

Bits	Description	Type	Reset
31:0	Read to sample the pad output enables (direction) PIO is currently driving to the GPIOs. On RP2040 there are 30 GPIOs, so the two most significant bits are hardwired to 0.	RO	0x00000000

PIO: DBG_CFGINFO Register

Offset: 0x044

Description

The PIO hardware has some free parameters that may vary between chip products. These should be provided in the chip datasheet, but are also exposed here.

Table 379.
DBG_CFGINFO Register

Bits	Description	Type	Reset
31:22	Reserved.	-	-
21:16	IMEM_SIZE : The size of the instruction memory, measured in units of one instruction	RO	-
15:12	Reserved.	-	-
11:8	SM_COUNT : The number of state machines this PIO instance is equipped with.	RO	-
7:6	Reserved.	-	-
5:0	FIFO_DEPTH : The depth of the state machine TX/RX FIFOs, measured in words. Joining fifos via SHIFTCTRL_FJOIN gives one FIFO with double this depth.	RO	-

PIO: INSTR_MEM0, INSTR_MEM1, ..., INSTR_MEM30, INSTR_MEM31 Registers

Offsets: 0x048, 0x04c, ..., 0x0c0, 0x0c4

位	描述	类型	复位值
31:0	<p>每个GPIO输入端均设有2级触发器同步器，以防止PIO逻辑发生亚稳态。这会增加输入延迟，对于高速同步IO（如SPI），可能需要绕过该同步器。该寄存器中的每个位对应一个GPIO。</p> <p>0 → 输入已同步（默认） 1 → 同步器已被绕过 如有疑问，请保持此寄存器全零。</p>	读写	0x00000000

PIO: DBG_PADOUT 寄存器

偏移量: 0x03c

表 377。
DBG_PADOUT 寄存器

位	描述	类型	复位值
31:0	读取以采样PIO当前驱动至GPIO的引脚输出值。RP2040共有30个GPIO，最高两位硬连线为0。	只读	0x00000000

PIO: DBG_PADOE 寄存器

偏移量: 0x040

表 378。
DBG_PADOE 寄存器

位	描述	类型	复位值
31:0	读取以采样PIO当前驱动至GPIO的引脚输出使能（方向）。RP2040共有30个GPIO，最高两位硬连线为0。	只读	0x00000000

PIO: DBG_CFGINFO 寄存器

偏移量: 0x044

描述

PIO 硬件包含某些自由参数，可能因芯片型号而异。

这些参数应在芯片数据手册中提供，但此处亦有展示。

表 379。
DBG_CFGINFO
寄存器

位	描述	类型	复位值
31:22	保留。	-	-
21:16	IMEM_SIZE : 指令存储器大小，单位为单条指令	只读	-
15:12	保留。	-	-
11:8	SM_COUNT : 该 PIO 实例所配置的状态机数量。	只读	-
7:6	保留。	-	-
5:0	FIFO_DEPTH : 状态机 TX/RX FIFO 的深度，单位为字。 通过 SHIFTCTRL_FJOIN 连接 FIFO，可形成深度加倍的单一 FIFO。	只读	-

PIO: INSTR_MEM0, INSTR_MEM1, ..., INSTR_MEM30, INSTR_MEM31 寄存器

偏移量: 0x048, 0x04C, ..., 0x0C0, 0x0C4

Table 380.
*INSTR_MEM0,
 INSTR_MEM1, ...,
 INSTR_MEM30,
 INSTR_MEM31
 Registers*

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Write-only access to instruction memory location N	WO	0x0000

PIO: SM0_CLKDIV, SM1_CLKDIV, SM2_CLKDIV, SM3_CLKDIV Registers

Offsets: 0x0c8, 0x0e0, 0x0f8, 0x110

Description

Clock divisor register for state machine N

Frequency = clock freq / (CLKDIV_INT + CLKDIV_FRAC / 256)

Table 381.
*SM0_CLKDIV,
 SM1_CLKDIV,
 SM2_CLKDIV,
 SM3_CLKDIV
 Registers*

Bits	Description	Type	Reset
31:16	INT: Effective frequency is sysclk/(int + frac/256). Value of 0 is interpreted as 65536. If INT is 0, FRAC must also be 0.	RW	0x0001
15:8	FRAC: Fractional part of clock divisor	RW	0x00
7:0	Reserved.	-	-

PIO: SM0_EXECCTRL, SM1_EXECCTRL, SM2_EXECCTRL, SM3_EXECCTRL Registers

Offsets: 0x0cc, 0x0e4, 0x0fc, 0x114

Description

Execution/behavioural settings for state machine N

Table 382.
*SM0_EXECCTRL,
 SM1_EXECCTRL,
 SM2_EXECCTRL,
 SM3_EXECCTRL
 Registers*

Bits	Description	Type	Reset
31	EXEC_STALLED: If 1, an instruction written to SMx_INSTR is stalled, and latched by the state machine. Will clear to 0 once this instruction completes.	RO	0x0
30	SIDE_EN: If 1, the MSB of the Delay/Side-set instruction field is used as side-set enable, rather than a side-set data bit. This allows instructions to perform side-set optionally, rather than on every instruction, but the maximum possible side-set width is reduced from 5 to 4. Note that the value of PINCTRL_SIDESET_COUNT is inclusive of this enable bit.	RW	0x0
29	SIDE_PINDIR: If 1, side-set data is asserted to pin directions, instead of pin values	RW	0x0
28:24	JMP_PIN: The GPIO number to use as condition for JMP PIN. Unaffected by input mapping.	RW	0x00
23:19	OUT_EN_SEL: Which data bit to use for inline OUT enable	RW	0x00
18	INLINE_OUT_EN: If 1, use a bit of OUT data as an auxiliary write enable When used in conjunction with OUT_STICKY, writes with an enable of 0 will deassert the latest pin write. This can create useful masking/override behaviour due to the priority ordering of state machine pin writes (SM0 < SM1 < ...)	RW	0x0
17	OUT_STICKY: Continuously assert the most recent OUT/SET to the pins	RW	0x0
16:12	WRAP_TOP: After reaching this address, execution is wrapped to wrap_bottom. If the instruction is a jump, and the jump condition is true, the jump takes priority.	RW	0x1f

表 380。
`INSTR_MEM0`,
`INSTR_MEM1`, ...,
`INSTR_MEM30`,
`INSTR_MEM31`
 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	对指令存储位置的只写访问 N	WO	0x0000

PIO: SM0_CLKDIV、SM1_CLKDIV、SM2_CLKDIV、SM3_CLKDIV 寄存器

偏移量: 0x0c8、0x0e0、0x0f8、0x110

描述

状态机的时钟分频寄存器 N

频率 = 时钟频率 / (CLKDIV_INT + CLKDIV_FRAC / 256)

表 381。
`SM0_CLKDIV`,
`SM1_CLKDIV`,
`SM2_CLKDIV`,
`SM3_CLKDIV`
 寄存器

位	描述	类型	复位值
31:16	INT : 有效频率为 sysclk/(int + frac/256)。 值为0时解释为65536。若 INT 为 0，则 FRAC 亦须为 0。	读写	0x0001
15:8	FRAC : 时钟分频的分数部分	读写	0x00
7:0	保留。	-	-

PIO: SM0_EXECCTRL、SM1_EXECCTRL、SM2_EXECCTRL、SM3_EXECCTRL 寄存器

偏移量: 0x0cc、0x0e4、0x0fc、0x114

描述

状态机的执行/行为设置 N

表 382。
`SM0_EXECCTRL`,
`SM1_EXECCTRL`,
`SM2_EXECCTRL`,
`SM3_EXECCTRL`
 寄存器

位	描述	类型	复位值
31	EXEC_STALLED : 若为1，则写入 <code>SMx_INSTR</code> 的指令被阻塞并由状态机锁存。该指令完成后将被清零。	只读	0x0
30	SIDE_EN : 若为1，Delay/Side-set 指令字段的最高有效位用作侧设使能位，而非侧设数据位。此设置允许指令选择性地执行侧设，而非每条指令均执行，但最大侧设宽度由5位减至4位。请注意， <code>PINCTRL_SIDESET_COUNT</code> 的值包含此使能位。	读写	0x0
29	SIDE_PINDIR : 若为1，侧设数据应用于引脚方向，而非引脚电平。	读写	0x0
28:24	JMP_PIN : 用于 <code>JMP PIN</code> 条件的 GPIO 编号，不受输入映射影响。	读写	0x00
23:19	OUT_EN_SEL : 选择用于内联OUT使能的数据位	读写	0x00
18	INLINE_OUT_EN : 若为1，则使用OUT数据中的某个位作为辅助写使能与OUT_STICKY同时使用时，使能为0的写操作将取消最近一次的引脚写入。这可以实现有效的掩码或覆盖行为 基于状态机引脚写入的优先级顺序 (<code>SM0 < SM1 < ...</code>)	读写	0x0
17	OUT_STICKY : 持续对引脚断言最近的OUT/SET信号	读写	0x0
16:12	WRAP_TOP : 达到此地址后，程序执行跳转至wrap_bottom。 若指令为跳转且跳转条件成立，则跳转具有优先权。	读写	0x1f

Bits	Description	Type	Reset
11:7	WRAP_BOTTOM : After reaching wrap_top, execution is wrapped to this address.	RW	0x00
6:5	Reserved.	-	-
4	STATUS_SEL : Comparison used for the MOV x, STATUS instruction. Enumerated values: 0x0 → TXLEVEL: All-ones if TX FIFO level < N, otherwise all-zeroes 0x1 → RXLEVEL: All-ones if RX FIFO level < N, otherwise all-zeroes	RW	0x0
3:0	STATUS_N : Comparison level for the MOV x, STATUS instruction	RW	0x0

PIO: SM0_SHIFTCTRL, SM1_SHIFTCTRL, SM2_SHIFTCTRL, SM3_SHIFTCTRL Registers

Offsets: 0x0d0, 0x0e8, 0x100, 0x118

Description

Control behaviour of the input/output shift registers for state machine N

Table 383.
SM0_SHIFTCTRL,
SM1_SHIFTCTRL,
SM2_SHIFTCTRL,
SM3_SHIFTCTRL
Registers

Bits	Description	Type	Reset
31	FJOIN_RX : When 1, RX FIFO steals the TX FIFO's storage, and becomes twice as deep. TX FIFO is disabled as a result (always reads as both full and empty). FIFOs are flushed when this bit is changed.	RW	0x0
30	FJOIN_TX : When 1, TX FIFO steals the RX FIFO's storage, and becomes twice as deep. RX FIFO is disabled as a result (always reads as both full and empty). FIFOs are flushed when this bit is changed.	RW	0x0
29:25	PULL_THRESH : Number of bits shifted out of OSR before autopull, or conditional pull (PULL IFEMPTY), will take place. Write 0 for value of 32.	RW	0x00
24:20	PUSH_THRESH : Number of bits shifted into ISR before autopush, or conditional push (PUSH IFFULL), will take place. Write 0 for value of 32.	RW	0x00
19	OUT_SHIFTDIR : 1 = shift out of output shift register to right. 0 = to left.	RW	0x1
18	IN_SHIFTDIR : 1 = shift input shift register to right (data enters from left). 0 = to left.	RW	0x1
17	AUTOPULL : Pull automatically when the output shift register is emptied, i.e. on or following an OUT instruction which causes the output shift counter to reach or exceed PULL_THRESH.	RW	0x0
16	AUTOPUSH : Push automatically when the input shift register is filled, i.e. on an IN instruction which causes the input shift counter to reach or exceed PUSH_THRESH.	RW	0x0
15:0	Reserved.	-	-

PIO: SM0_ADDR, SM1_ADDR, SM2_ADDR, SM3_ADDR Registers

Offsets: 0x0d4, 0x0ec, 0x104, 0x11c

位	描述	类型	复位值
11:7	WRAP_BOTTOM : 达到wrap_top后，程序执行跳转至此地址。	读写	0x00
6:5	保留。	-	-
4	STATUS_SEL : 用于MOV x, STATUS指令中的比较选择。	读写	0x0
	枚举值:		
	0x0 → TXLEVEL: 当TX FIFO级别 < N时，返回全1，否则返回全0		
	0x1 → RXLEVEL: 当RX FIFO级别 < N时，返回全1，否则返回全0		
3:0	STATUS_N : MOV x, STATUS 指令的比较等级	读写	0x0

PIO: SM0_SHIFTCTRL, SM1_SHIFTCTRL, SM2_SHIFTCTRL, SM3_SHIFTCTRL 寄存器

偏移量: 0x0d0, 0x0e8, 0x100, 0x118

描述

控制状态机 N 输入/输出移位寄存器的行为

表383。
SM0_SHIFTCTRL,
SM1_SHIFTCTRL,
SM2_SHIFTCTRL,
SM3_SHIFTCTRL
寄存器

位	描述	类型	复位值
31	FJOIN_RX : 当该位为1时，接收FIFO占用发送FIFO的存储空间，深度加倍。 因此，发送FIFO被禁用（始终读作既满又空）。 该位改变时，FIFO将被清空。	读写	0x0
30	FJOIN_TX : 当该位为1时，发送FIFO占用接收FIFO的存储空间，深度加倍。 因此，接收FIFO被禁用（始终读作既满又空）。 该位改变时，FIFO将被清空。	读写	0x0
29:25	PULL_THRESH : 在自动拉取或条件拉取（空时拉取）发生前，从OS R中移出的位数。 写入值0表示32。	读写	0x00
24:20	PUSH_THRESH : 在执行自动推送（autopush）或条件推送（PUSH IFFULL）前，移入ISR的比特数。 写入值0表示32。	读写	0x00
19	OUT_SHIFTDIR : 1 = 从输出移位寄存器向右移位；0 = 向左移位。	读写	0x1
18	IN_SHIFTDIR : 1 = 输入移位寄存器向右移位（数据从左侧进入）；0 = 向左移位。	读写	0x1
17	AUTOPULL : 当输出移位寄存器清空时自动拉取，即在执行OUT指令且输出移位计数器达到或超过PULL_THRESH时。	读写	0x0
16	AUTOPUSH : 当输入移位寄存器填满时自动推送，即在执行IN指令导致输入移位计数器达到或超过PUSH_THRESH时。	读写	0x0
15:0	保留。	-	-

PIO: SM0_ADDR、SM1_ADDR、SM2_ADDR、SM3_ADDR寄存器

偏移: 0x0d4、0x0ec、0x104、0x11c

Table 384. SMO_ADDR,
SM1_ADDR,
SM2_ADDR,
SM3_ADDR Registers

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	Current instruction address of state machine N	RO	0x00

PIO: SMO_INSTR, SM1_INSTR, SM2_INSTR, SM3_INSTR Registers

Offsets: 0x0d8, 0x0f0, 0x108, 0x120

Table 385.
SMO_INSTR,
SM1_INSTR,
SM2_INSTR,
SM3_INSTR Registers

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Read to see the instruction currently addressed by state machine N 's program counter. Write to execute an instruction immediately (including jumps) and then resume execution.	RW	-

PIO: SMO_PINCTRL, SM1_PINCTRL, SM2_PINCTRL, SM3_PINCTRL Registers

Offsets: 0x0dc, 0x0f4, 0x10c, 0x124

Description

State machine pin control

Table 386.
SMO_PINCTRL,
SM1_PINCTRL,
SM2_PINCTRL,
SM3_PINCTRL Registers

Bits	Description	Type	Reset
31:29	SIDESET_COUNT: The number of MSBs of the Delay/Side-set instruction field which are used for side-set. Inclusive of the enable bit, if present. Minimum of 0 (all delay bits, no side-set) and maximum of 5 (all side-set, no delay).	RW	0x0
28:26	SET_COUNT: The number of pins asserted by a SET. In the range 0 to 5 inclusive.	RW	0x5
25:20	OUT_COUNT: The number of pins asserted by an OUT PINS, OUT PINDIRS or MOV PINS instruction. In the range 0 to 32 inclusive.	RW	0x00
19:15	IN_BASE: The pin which is mapped to the least-significant bit of a state machine's IN data bus. Higher-numbered pins are mapped to consecutively more-significant data bits, with a modulo of 32 applied to pin number.	RW	0x00
14:10	SIDESET_BASE: The lowest-numbered pin that will be affected by a side-set operation. The MSBs of an instruction's side-set/delay field (up to 5, determined by SIDESET_COUNT) are used for side-set data, with the remaining LSBs used for delay. The least-significant bit of the side-set portion is the bit written to this pin, with more-significant bits written to higher-numbered pins.	RW	0x00
9:5	SET_BASE: The lowest-numbered pin that will be affected by a SET PINS or SET PINDIRS instruction. The data written to this pin is the least-significant bit of the SET data.	RW	0x00
4:0	OUT_BASE: The lowest-numbered pin that will be affected by an OUT PINS, OUT PINDIRS or MOV PINS instruction. The data written to this pin will always be the least-significant bit of the OUT or MOV data.	RW	0x00

PIO: INTR Register

Offset: 0x128

Description

Raw Interrupts

表384。SM0_ADDR,
SM1_ADDR,
SM2_ADDR,
SM3_ADDR寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	状态机当前指令地址 N	只读	0x00

PIO: SM0_INSTR, SM1_INSTR, SM2_INSTR, SM3_INSTR寄存器

偏移量: 0x0d8, 0x0f0, 0x108, 0x120

表385。
SM0_INSTR,
SM1_INSTR,
SM2_INSTR,
SM3_INSTR寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	读取表示状态机 N程序计数器当前指向的指令。 写入以立即执行指令（包括跳转）并随后恢复执行。	读写	-

PIO: SM0_PINCTRL, SM1_PINCTRL, SM2_PINCTRL, SM3_PINCTRL寄存器

偏移量: 0x0dc, 0x0f4, 0x10c, 0x124

描述

状态机引脚控制

表386。
SM0_PINCTRL,
SM1_PINCTRL,
SM2_PINCTRL,
SM3_PINCTRL寄存器

位	描述	类型	复位值
31:29	SIDESET_COUNT: 用于side-set的延迟/side-set指令字段最高有效位数。包括启用位（如存在）。最小为0（所有为延迟位，无侧置），最大为5（所有为侧置，无延迟）。	读写	0x0
28:26	SET_COUNT: 由SET断言的引脚数量，范围为0至5（含）。	读写	0x5
25:20	OUT_COUNT: 由OUT PINS、OUT PINDIRS或MOV PINS指令断言的引脚数量。范围为0至32（含）。	读写	0x00
19:15	IN_BASE: 映射至状态机IN数据总线最低有效位的引脚编号。编号较高的引脚依次映射至更高有效位，且对引脚编号取模32。	读写	0x00
14:10	SIDESET_BASE: 受侧置操作影响的最低编号引脚。指令侧置/延迟字段的最高有效位（最多5位，由SIDESET_COUNT确定）用于侧置数据，其余最低有效位用于延迟计数。side-set部分的最低有效位写入该引脚，更高有效位写入编号更高的引脚。	读写	0x00
9:5	SET_BASE: 将被SET PINS或SET PINDIRS指令影响的编号最低的引脚。写入该引脚的数据为SET数据的最低有效位。	读写	0x00
4:0	OUT_BASE: 将被OUT PINS、OUT PINDIRS或MOV PINS指令影响的编号最低的引脚。写入该引脚的数据始终为OUT或MOV数据的最低有效位。	读写	0x00

PIO: INTR 寄存器

偏移: 0x128

描述

原始中断

Table 387. INTR Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RO	0x0
10	SM2	RO	0x0
9	SM1	RO	0x0
8	SM0	RO	0x0
7	SM3_TXNFULL	RO	0x0
6	SM2_TXNFULL	RO	0x0
5	SM1_TXNFULL	RO	0x0
4	SM0_TXNFULL	RO	0x0
3	SM3_RXNEMPTY	RO	0x0
2	SM2_RXNEMPTY	RO	0x0
1	SM1_RXNEMPTY	RO	0x0
0	SM0_RXNEMPTY	RO	0x0

PIO: IRQ0_INTE Register

Offset: 0x12c

Description

Interrupt Enable for irq0

Table 388. IRQ0_INTE Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RW	0x0
10	SM2	RW	0x0
9	SM1	RW	0x0
8	SM0	RW	0x0
7	SM3_TXNFULL	RW	0x0
6	SM2_TXNFULL	RW	0x0
5	SM1_TXNFULL	RW	0x0
4	SM0_TXNFULL	RW	0x0
3	SM3_RXNEMPTY	RW	0x0
2	SM2_RXNEMPTY	RW	0x0
1	SM1_RXNEMPTY	RW	0x0
0	SM0_RXNEMPTY	RW	0x0

PIO: IRQ0_INTF Register

Offset: 0x130

Description

Interrupt Force for irq0

表 387. INTR
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

PIO：IRQ0_INTE 寄存器

偏移: 0x12c

描述

irq0中断使能

表 388. IRQ0_INTE
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ0_INTF 寄存器

偏移: 0x130

描述

irq0中断强制

Table 389. IRQ0_INTF Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RW	0x0
10	SM2	RW	0x0
9	SM1	RW	0x0
8	SM0	RW	0x0
7	SM3_TXNFULL	RW	0x0
6	SM2_TXNFULL	RW	0x0
5	SM1_TXNFULL	RW	0x0
4	SM0_TXNFULL	RW	0x0
3	SM3_RXNEMPTY	RW	0x0
2	SM2_RXNEMPTY	RW	0x0
1	SM1_RXNEMPTY	RW	0x0
0	SM0_RXNEMPTY	RW	0x0

PIO: IRQ0_INTS Register

Offset: 0x134

Description

Interrupt status after masking & forcing for irq0

Table 390. IRQ0_INTS Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RO	0x0
10	SM2	RO	0x0
9	SM1	RO	0x0
8	SM0	RO	0x0
7	SM3_TXNFULL	RO	0x0
6	SM2_TXNFULL	RO	0x0
5	SM1_TXNFULL	RO	0x0
4	SM0_TXNFULL	RO	0x0
3	SM3_RXNEMPTY	RO	0x0
2	SM2_RXNEMPTY	RO	0x0
1	SM1_RXNEMPTY	RO	0x0
0	SM0_RXNEMPTY	RO	0x0

PIO: IRQ1_INTE Register

Offset: 0x138

Description

Interrupt Enable for irq1

表389. IRQ0_INTF
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO: IRQ0_INTS 寄存器

偏移量: 0x134

说明

irq0中断屏蔽及强制后的状态

表390. IRQ0_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

PIO: IRQ1_INTE 寄存器

偏移: 0x138

描述

irq1中断使能

Table 391. IRQ1_INTE Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RW	0x0
10	SM2	RW	0x0
9	SM1	RW	0x0
8	SM0	RW	0x0
7	SM3_TXNFULL	RW	0x0
6	SM2_TXNFULL	RW	0x0
5	SM1_TXNFULL	RW	0x0
4	SM0_TXNFULL	RW	0x0
3	SM3_RXNEMPTY	RW	0x0
2	SM2_RXNEMPTY	RW	0x0
1	SM1_RXNEMPTY	RW	0x0
0	SM0_RXNEMPTY	RW	0x0

PIO: IRQ1_INTF Register

Offset: 0x13c

Description

Interrupt Force for irq1

Table 392. IRQ1_INTF Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RW	0x0
10	SM2	RW	0x0
9	SM1	RW	0x0
8	SM0	RW	0x0
7	SM3_TXNFULL	RW	0x0
6	SM2_TXNFULL	RW	0x0
5	SM1_TXNFULL	RW	0x0
4	SM0_TXNFULL	RW	0x0
3	SM3_RXNEMPTY	RW	0x0
2	SM2_RXNEMPTY	RW	0x0
1	SM1_RXNEMPTY	RW	0x0
0	SM0_RXNEMPTY	RW	0x0

PIO: IRQ1_INTS Register

Offset: 0x140

Description

Interrupt status after masking & forcing for irq1

表391. IRQ1_INTF
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ1_INTF 寄存器

偏移: 0x13c

描述

irq1中断强制

表392. IRQ1_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	读写	0x0
10	SM2	读写	0x0
9	SM1	读写	0x0
8	SM0	读写	0x0
7	SM3_TXNFULL	读写	0x0
6	SM2_TXNFULL	读写	0x0
5	SM1_TXNFULL	读写	0x0
4	SM0_TXNFULL	读写	0x0
3	SM3_RXNEMPTY	读写	0x0
2	SM2_RXNEMPTY	读写	0x0
1	SM1_RXNEMPTY	读写	0x0
0	SM0_RXNEMPTY	读写	0x0

PIO：IRQ1_INTS 寄存器

偏移: 0x140

说明

irq1中断屏蔽及强制后的状态

Table 393. IRQ1_INTS Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	SM3	RO	0x0
10	SM2	RO	0x0
9	SM1	RO	0x0
8	SM0	RO	0x0
7	SM3_TXNFULL	RO	0x0
6	SM2_TXNFULL	RO	0x0
5	SM1_TXNFULL	RO	0x0
4	SM0_TXNFULL	RO	0x0
3	SM3_RXNEMPTY	RO	0x0
2	SM2_RXNEMPTY	RO	0x0
1	SM1_RXNEMPTY	RO	0x0
0	SM0_RXNEMPTY	RO	0x0

表393. IRQ1_INTS
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	SM3	只读	0x0
10	SM2	只读	0x0
9	SM1	只读	0x0
8	SM0	只读	0x0
7	SM3_TXNFULL	只读	0x0
6	SM2_TXNFULL	只读	0x0
5	SM1_TXNFULL	只读	0x0
4	SM0_TXNFULL	只读	0x0
3	SM3_RXNEMPTY	只读	0x0
2	SM2_RXNEMPTY	只读	0x0
1	SM1_RXNEMPTY	只读	0x0
0	SM0_RXNEMPTY	只读	0x0

Chapter 4. Peripherals

4.1. USB

4.1.1. Overview

Prerequisite Knowledge Required

This section requires knowledge of the USB protocol. We recommend [USB Made Simple](#) if you are unclear on the terminology used in this section.

RP2040 contains a USB 2.0 controller that can operate as either:

- a Full Speed device (12Mbps)
- a host that can communicate with both Low Speed (1.5Mbps) and Full Speed devices. This includes multiple downstream devices connected to a USB hub.

There is an integrated USB 1.1 PHY which interfaces the USB controller with the **DP** and **DM** pins of the chip.

4.1.1.1. Features

The USB controller hardware handles the low level USB protocol, meaning the main job of the programmer is to configure the controller and then provide / consume data buffers in response to events on the bus. The controller interrupts the processor when it needs attention. The USB controller has 4kB of DPSRAM which is used for configuration and data buffers.

4.1.1.1.1. Device Mode

- USB 2.0-compatible Full Speed device (12Mbps)
- Supports up to 32 endpoints (Endpoints 0 → 15 in both in and out directions)
- Supports **Control**, **Isochronous**, **Bulk**, and **Interrupt** endpoint types
- Supports double buffering
- 3840 bytes of usable buffer space in DPSRAM. This is equivalent to 60×64 -byte buffers.

4.1.1.1.2. Host Mode

- Can communicate with Full Speed (12Mbps) devices and Low Speed devices (1.5Mbps)
- Can communicate with multiple devices via a USB hub, including Low Speed devices connected to a Full Speed hub
- Can poll up to 15 interrupt endpoints in hardware. (Interrupt endpoints are used by hubs to notify the host of connect/disconnect events, mice to notify the host of movement etc.)

第4章 外设

4.1. USB

4.1.1. 概述

先决知识

本节内容需要具备USB协议相关知识。若您对本节所用术语不熟悉，建议参考《USB Made Simple》一书。

RP2040内置USB 2.0控制器，支持下列任一模式运行：

- 全速设备（12Mbps）
- 能够与低速（1.5Mbps）和全速设备通信的主机。包括连接到USB集线器的多个下游设备。

集成了USB 1.1 PHY，用于将USB控制器与芯片的 DP 和 DM 引脚接口连接。

4.1.1.1. 功能特性

USB控制器硬件负责处理底层USB协议，程序员的主要任务是配置控制器，并在总线事件发生时提供或使用数据缓冲区。当控制器需要处理器响应时，会触发中断。USB控制器配备4kB的DPSRAM，用于配置和数据缓冲。

4.1.1.1.1. 设备模式

- 兼容USB 2.0的全速设备（12Mbps）
- 支持最多32个端点（端点0 → 15，含输入和输出方向）
- 支持控制、等时、批量和中断端点类型
- 支持双缓冲
- DPSRAM中可用缓冲区空间为3840字节，相当于60个64字节的缓冲区。

4.1.1.1.2. 主机模式

- 能够与全速（12Mbps）及低速（1.5Mbps）设备通信
- 能够通过USB集线器与多个设备通信，包括连接至全速集线器的低速设备
- 硬件支持轮询最多15个中断端点。（中断端点用于集线器通知主机连接/断开事件，鼠标通知主机移动等。）

4.1.2. Architecture

4.1.2.1. Clock speed

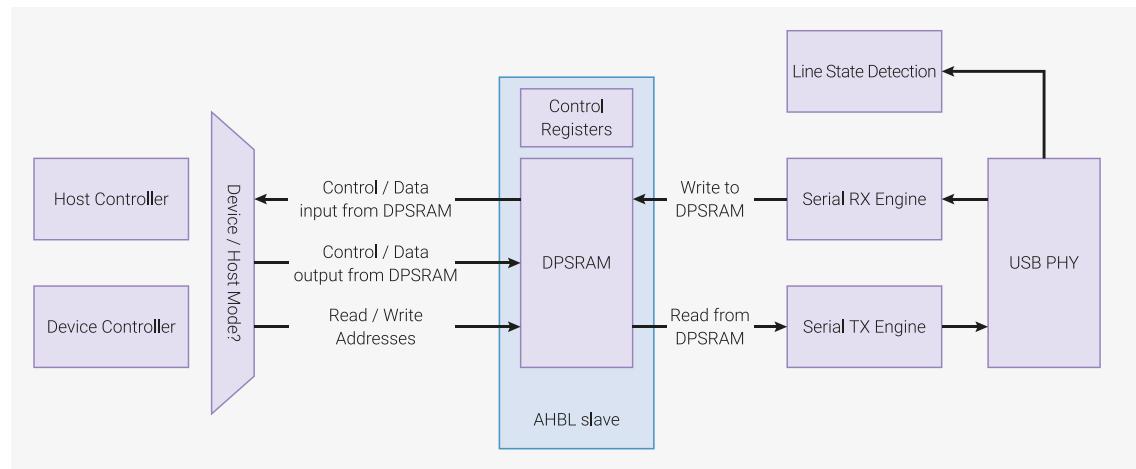
This controller requires `clk_usb` to be running at 48MHz.

NOTE

`clk_sys` must also be running at > 48MHz. See [RP2040-E16](#).

4.1.2.2. Overview

Figure 57. A simplified overview of the USB controller architecture.



The USB controller is an area efficient design that muxes a device controller or host controller onto a common set of components. Each component is detailed below.

4.1.2.3. USB PHY

The USB PHY provides the electrical interface between the USB `DP` and `DM` pins and the digital logic of the controller. The `DP` and `DM` pins are a differential pair, meaning the values are always the inverse of each other, except to encode a specific line state (`SE0`, etc). The USB PHY drives the `DP` and `DM` pins to transmit data, as well as performing a differential receive of any incoming data. The USB PHY provides both single-ended and differential receive data to the line state detection module.

The USB PHY has built in pull-up and pull-down resistors. If the controller is acting as a Full Speed device then the `DP` pin is pulled up to indicate to the host that a Full Speed device has been connected. In host mode, a weak pull down is applied to `DP` and `DM` so that the lines are pulled to a logical zero until the device pulls up `DP` for Full Speed or `DM` for Low Speed.

4.1.2.4. Line state detection

The [2] defines several line states (Bus Reset, Connected, Suspend, Resume, Data 1, Data 0, etc) that need to be detected. The line state detection module has several state machines to detect these states and signal events to the other hardware components. There is no shared clock signal in USB, so the RX data must be sampled by an internal clock. The maximum data rate of USB Full Speed is 12Mbps. The RX data is sampled at 48MHz, giving 4 clock cycles to capture and filter the bus state. The line state detection module distributes the filtered RX data to the Serial RX Engine.

4.1.2. 架构

4.1.2.1. 时钟频率

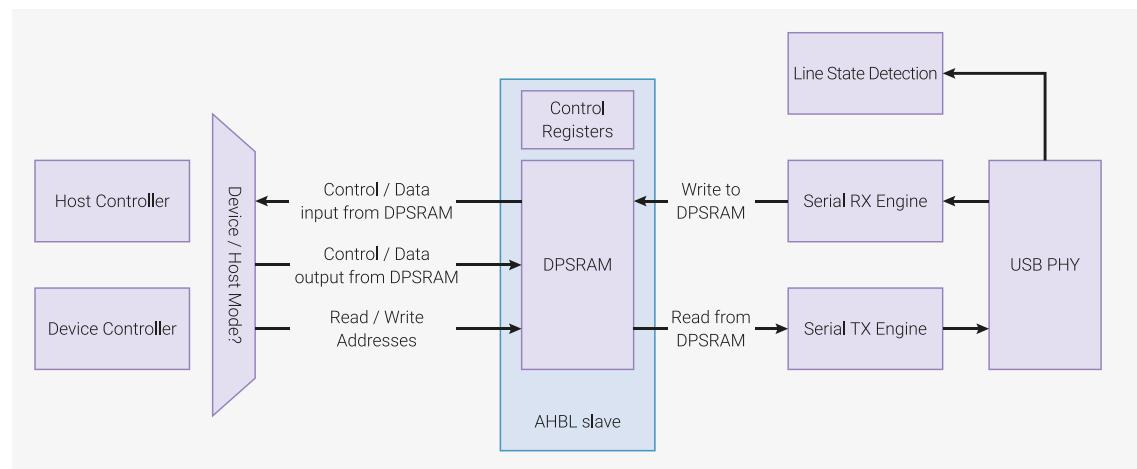
该控制器要求 `clk_usb` 以48MHz运行。

注意

`clk_sys` 也必须运行于高于48MHz的频率。详见RP2040-E16。

4.1.2.2. 概述

图57。USB控制器架构的简化概述
。



USB控制器是一种高效节省空间的设计，将设备控制器或主机控制器复用在一组通用组件上。各组件详述如下。

4.1.2.3. USB PHY

USB PHY 提供 USB `DP`与 `DM`引脚和控制器数字逻辑之间的电气接口。`DP`与 `DM`引脚成差分对，意味着其数值始终相反，除非用于编码特定线路状态（如`SE0`等）。USB PHY 驱动 `DP`与 `DM`引脚以传输数据，同时负责对任何输入数据进行差分接收。USB PHY 向线路状态检测模块提供单端和差分接收的数据。

USB PHY 内置上拉电阻与下拉电阻。若控制器作为全速设备，`DP`引脚会被上拉，以向主机指示已连接全速设备。在主机模式下，`DP`与 `DM`引脚施加弱下拉，使线路保持逻辑低电平，直到设备上拉 `DP`以表示全速，或上拉 `DM`以表示低速。

4.1.2.4. 线路状态检测

[2] 定义了若干需要检测的线路状态（总线复位、已连接、挂起、恢复、数据1、数据0等）。线路状态检测模块配备多个状态机，用于检测这些状态并向其他硬件组件发送事件信号。USB 中不存在共享时钟信号，因此接收数据必须由内部时钟采样。USB 全速的最大数据传输速率为 12Mbps。接收数据以 48MHz 频率采样，提供 4 个时钟周期用于捕获和滤波总线状态。线路状态检测模块将滤波后的接收数据分发至串行接收引擎。

4.1.2.5. Serial RX Engine

The serial receive engine decodes receive data captured by the line state detection module. It produces the following information:

- The **PID** of the incoming data packet
- The device address for the incoming data
- The device endpoint for the incoming data
- Data bytes

The serial receive engine also detects errors in RX data by performing a CRC check on the incoming data. Any errors are signalled to the other hardware blocks and can raise an interrupt.

i NOTE

If you disconnect the USB cable during a packet in either host or device mode you will see errors raised by the hardware. Your software will need to take this scenario into account if you enable error interrupts.

4.1.2.6. Serial TX Engine

The serial transmit engine is a mirror of the serial receive engine. It is connected to the currently active controller (either device or host). It creates **TOKEN** and **DATA** packets, including calculating the CRC, and transmits them on the bus.

4.1.2.7. DPSRAM

The USB controller has 4kB (4096 bytes) of DPSRAM (Dual Port SRAM). The DPSRAM is used to store control registers and data buffers. The DPSRAM is accessible as a 32-bit wide memory at address 0 of the USB controller (**0x50100000**).

The DPSRAM has the following characteristics, which are different to most registers on RP2040:

- Supports 8/16/32-bit accesses. Registers typically support 32-bit accesses only
- The DPSRAM does not support set / clear aliases. RP2040 registers typically support these

Data Buffers are typically 64 bytes long as this is the max normal packet size for most FS packets. For Isochronous endpoints a maximum buffer size of 1023 bytes is supported. For other packet types the maximum size is 64 bytes per buffer.

4.1.2.7.1. Concurrent access

The DPSRAM in the USB controller should be considered asynchronous and not atomic. It is a dual port SRAM which means the processor has a port to read/write the memory and the USB controller also has a port to read/write the memory. This means that both the processor and the USB controller can access the same memory address at the same time. One could be writing and one could be reading. It is possible to get inconsistent data if the controller is reading the memory while the processor is writing the memory. Care must be taken to avoid this scenario.

The **AVAILABLE** bit in the buffer control register is used to indicate who has ownership of a buffer. This bit should be set to 1 by the processor to give the controller ownership of the buffer. The controller will set the bit back to 0 when it has used the buffer. The **AVAILABLE** bit should be set separately to the rest of the data in the buffer control register, so that the rest of the data in the buffer control register is accurate when the **AVAILABLE** bit is set.

This is necessary because the processor clock **clk_sys** can be running several times faster than the **clk_usb** clock. Therefore **clk_sys** can update the data **during** a read by the USB controller on a slower clock. The correct process is:

- Write buffer information (length, etc.) to buffer control register

4.1.2.5. 串行接收引擎

串行接收引擎对线路状态检测模块捕获的接收数据进行解码。其输出以下信息：

- 传入数据包的 **PID**
- 传入数据的设备地址
- 传入数据的设备端点
- 数据字节

串行接收引擎通过对传入数据执行CRC校验，检测RX数据中的错误。任何错误都会向其他硬件模块发出信号，并可能触发中断。

i 注意

如果在主机模式或设备模式的数据包传输过程中断开USB线缆，硬件会报告错误。若启用错误中断，您的软件必须考虑该场景。

4.1.2.6 串行发送引擎

串行发送引擎是串行接收引擎的镜像。它连接到当前活动的控制器（设备端或主机端）。它生成 **TOKEN** 和 **DATA** 数据包，包含CRC计算，并在总线上发送这些数据包。

4.1.2.7 DPSRAM

USB控制器配备4KB（4096字节）DPSRAM（双端口静态随机存取存储器）。DPSRAM用于存储控制寄存器和数据缓冲区。DPSRAM可作为32位宽内存，从USB控制器地址0（**0x5010000**）访问。

DPSRAM具有以下特性，这些特性与RP2040上的大多数寄存器不同：

- 支持8位、16位和32位访问。寄存器通常仅支持32位访问。
- DPSRAM不支持设置/清除别名功能。RP2040寄存器通常支持这些功能。

数据缓冲区通常为64字节，这是大多数FS数据包的最大常规包大小。对于等时端点，支持最大1023字节的缓冲区大小。对于其他数据包类型，每个缓冲区的最大大小为64字节。

4.1.2.7.1. 并发访问

USB控制器中的DPSRAM应视为异步且非原子操作。它是一种双端口SRAM，这意味着处理器拥有一个端口用于读写内存，USB控制器也拥有一个端口用于读写内存。这意味着处理器和USB控制器可以同时访问相同的内存地址。一方可能正在写入，另一方可能正在读取。如果控制器在处理器写入内存时读取内存，可能会导致数据不一致。必须谨慎以避免此类情况发生。

缓冲区控制寄存器中的 **AVAILABLE** 位用于指示缓冲区的所有权归属。处理器应将该位设置为1，以将缓冲区所有权交予控制器。控制器在使用缓冲区后，会将该位重新置0。应将 **AVAILABLE** 位与缓冲区控制寄存器中的其他数据分开设置，以确保在设置 **AVAILABLE** 位时，缓冲区控制寄存器中的其余数据保持准确。

这是因为处理器时钟 **clk_sys** 可能远快于 **clk_usb** 时钟。

因此，**clk_sys** 可能在USB控制器以较慢时钟读取数据时更新数据。正确的操作流程是：

- 将缓冲区信息（长度等）写入缓冲区控制寄存器

- `nop` for some `clk_sys` cycles to ensure that at least one `clk_usb` cycle has passed. For example if `clk_sys` was running at 125MHz and `clk_usb` was running at 48MHz then 125/48 rounded up would be 3 `nop` instructions
- Set `AVAILABLE` bit

If `clk_sys` and `clk_usb` are running at the same frequency then it is not necessary to set the `AVAILABLE` bit separately.

NOTE

When the controller is writing status back to the DPSRAM it does a 16 bit write to the lower 2 bytes for buffer 0 and the upper 2 bytes for buffer 1. Therefore, if using double buffered mode, it is safest to treat the buffer control register as two 16 bit registers when updating it in software.

4.1.2.7.2. Layout

Addresses `0x0-0xff` are used for control registers containing configuration data. The remaining space, addresses `0x100-0xffff` (3840 bytes) can be used for data buffers. The controller has control registers that start at address `0x10000`.

The memory layout is different depending on if the controller is in Device or Host mode. In device mode, there are multiple endpoints a host can access so there must be endpoint control and buffer control registers for each endpoint. In host mode, the host software running on the processor is deciding which endpoints and which devices to access, so there only needs to be one set of endpoint control and buffer control registers. As well as software driven transfers, the host controller can poll up to 15 interrupt endpoints and has a register for each of these interrupt endpoints.

Table 394. DPSRAM layout

Offset	Device Function	Host Function
0x0	Setup packet (8 bytes)	
0x8	EP1 in control	Interrupt endpoint control 1
0xc	EP1 out control	Spare
0x10	EP2 in control	Interrupt endpoint control 2
0x14	EP2 out control	Spare
0x18	EP3 in control	Interrupt endpoint control 3
0x1c	EP3 out control	Spare
0x20	EP4 in control	Interrupt endpoint control 4
0x24	EP4 out control	Spare
0x28	EP5 in control	Interrupt endpoint control 5
0x2c	EP5 out control	Spare
0x30	EP6 in control	Interrupt endpoint control 6
0x34	EP6 out control	Spare
0x38	EP7 in control	Interrupt endpoint control 7
0x3c	EP7 out control	Spare
0x40	EP8 in control	Interrupt endpoint control 8
0x44	EP8 out control	Spare
0x48	EP9 in control	Interrupt endpoint control 9
0x4c	EP9 out control	Spare
0x50	EP10 in control	Interrupt endpoint control 10
0x54	EP10 out control	Spare

- 执行若干 `nop` 指令，经过 `clk_sys` 周期，以确保至少经过一个 `clk_usb` 周期。例如，若 `clk_sys` 运行于 125MHz，`clk_usb` 运行于 48MHz，则 125/48 向上取整即为 3 条 `nop` 指令。

- 设置 AVAILABLE 位**

若 `clk_sys` 与 `clk_usb` 频率相同，则无需单独设置 `AVAILABLE` 位。

i 注意

当控制器向 DPSRAM 回写状态时，会对缓冲区 0 的低两字节和缓冲区 1 的高两字节执行 16 位写操作。因此，若使用双缓冲模式，软件更新缓冲区控制寄存器时，将其视为两个 16 位寄存器处理最为安全。

4.1.2.7.2. 布局

地址 `0x0` 至 `0xff` 区域用于存储包含配置信息的控制寄存器。剩余地址空间 `0x100` 至 `0xffff`（共 3840 字节）可用作数据缓冲区。控制器具有从地址 `0x10000` 开始的控制寄存器。

内存布局根据控制器是处于设备模式还是主机模式而不同。在设备模式下，主机可以访问多个端点，因此每个端点必须配置端点控制寄存器和缓冲区控制寄存器。

在主机模式下，运行于处理器上的主机软件负责决定访问哪些端点和设备，因此仅需一组端点控制寄存器和缓冲区控制寄存器。除软件驱动的传输外，主机控制器可轮询最多 15 个中断端点，并为每个中断端点配备相应寄存器。

表394. DPSRAM
布局

偏移量	设备功能	主机功能
0x0	设置包（8字节）	
0x8	EP1输入控制	中断端点控制1
0xc	EP1输出控制	备用
0x10	EP2输入控制	中断端点控制2
0x14	EP2输出控制	备用
0x18	EP3输入控制	中断端点控制3
0x1c	EP3输出控制	备用
0x20	EP4 输入控制	中断端点控制 4
0x24	EP4 输出控制	备用
0x28	EP5 输入控制	中断端点控制 5
0x2c	EP5 输出控制	备用
0x30	EP6 输入控制	中断端点控制 6
0x34	EP6 输出控制	备用
0x38	EP7 输入控制	中断端点控制 7
0x3c	EP7 输出控制	备用
0x40	EP8 输入控制	中断端点控制 8
0x44	EP8 输出控制	备用
0x48	EP9 输入控制	中断端点控制 9
0x4c	EP9 出端控制	备用
0x50	EP10 入端控制	中断端点控制 10
0x54	EP10 出端控制	备用

Offset	Device Function	Host Function
0x58	EP11 in control	Interrupt endpoint control 11
0x5c	EP11 out control	Spare
0x60	EP12 in control	Interrupt endpoint control 12
0x64	EP12 out control	Spare
0x68	EP13 in control	Interrupt endpoint control 13
0x6c	EP13 out control	Spare
0x70	EP14 in control	Interrupt endpoint control 14
0x74	EP14 out control	Spare
0x78	EP15 in control	Interrupt endpoint control 15
0x7c	EP15 out control	Spare
0x80	EP0 in buffer control	EPx buffer control
0x84	EP0 out buffer control	Spare
0x88	EP1 in buffer control	Interrupt endpoint buffer control 1
0x8c	EP1 out buffer control	Spare
0x90	EP2 in buffer control	Interrupt endpoint buffer control 2
0x94	EP2 out buffer control	Spare
0x98	EP3 in buffer control	Interrupt endpoint buffer control 3
0x9c	EP3 out buffer control	Spare
0xa0	EP4 in buffer control	Interrupt endpoint buffer control 4
0xa4	EP4 out buffer control	Spare
0xa8	EP5 in buffer control	Interrupt endpoint buffer control 5
0xac	EP5 out buffer control	Spare
0xb0	EP6 in buffer control	Interrupt endpoint buffer control 6
0xb4	EP6 out buffer control	Spare
0xb8	EP7 in buffer control	Interrupt endpoint buffer control 7
0xbc	EP7 out buffer control	Spare
0xc0	EP8 in buffer control	Interrupt endpoint buffer control 8
0xc4	EP8 out buffer control	Spare
0xc8	EP9 in buffer control	Interrupt endpoint buffer control 9
0xcc	EP9 out buffer control	Spare
0xd0	EP10 in buffer control	Interrupt endpoint buffer control 10
0xd4	EP10 out buffer control	Spare
0xd8	EP11 in buffer control	Interrupt endpoint buffer control 11
0xdc	EP11 out buffer control	Spare
0xe0	EP12 in buffer control	Interrupt endpoint buffer control 12
0xe4	EP12 out buffer control	Spare

偏移量	设备功能	主机功能
0x58	EP11 入端控制	中断端点控制 11
0x5c	EP11 出端控制	备用
0x60	EP12 入端控制	中断端点控制 12
0x64	EP12 出端控制	备用
0x68	EP13 入端控制	中断端点控制 13
0x6c	EP13 出端控制	备用
0x70	EP14 入端控制	中断端点控制 14
0x74	EP14 出端控制	备用
0x78	EP15 入端控制	中断端点控制 15
0x7c	EP15 出口控制	备用
0x80	EP0 输入缓冲控制	EPx 缓冲控制
0x84	EP0 出口缓冲控制	备用
0x88	EP1 输入缓冲控制	中断端点缓冲控制 1
0x8c	EP1 出口缓冲控制	备用
0x90	EP2 输入缓冲控制	中断端点缓冲控制 2
0x94	EP2 出口缓冲控制	备用
0x98	EP3 输入缓冲控制	中断端点缓冲控制 3
0x9c	EP3 出口缓冲控制	备用
0xa0	EP4 输入缓冲控制	中断端点缓冲控制 4
0xa4	EP4 出口缓冲控制	备用
0xa8	EP5 输入缓冲控制	中断端点缓冲控制 5
0xac	EP5 出端点缓冲区控制	备用
0xb0	EP6 入端点缓冲区控制	中断端点缓冲区控制 6
0xb4	EP6 出端点缓冲区控制	备用
0xb8	EP7 入端点缓冲区控制	中断端点缓冲区控制 7
0xbc	EP7 出端点缓冲区控制	备用
0xc0	EP8 入端点缓冲区控制	中断端点缓冲区控制 8
0xc4	EP8 出端点缓冲区控制	备用
0xc8	EP9 入端点缓冲区控制	中断端点缓冲区控制 9
0xcc	EP9 出端点缓冲区控制	备用
0xd0	EP10 入端点缓冲区控制	中断端点缓冲区控制 10
0xd4	EP10 输出缓冲区控制	备用
0xd8	EP11 输入缓冲区控制	中断端点缓冲区控制 11
0xdc	EP11 输出缓冲区控制	备用
0xe0	EP12 输入缓冲区控制	中断端点缓冲区控制 12
0xe4	EP12 输出缓冲区控制	备用

Offset	Device Function	Host Function
0xe8	EP13 in buffer control	Interrupt endpoint buffer control 13
0xec	EP13 out buffer control	Spare
0xf0	EP14 in buffer control	Interrupt endpoint buffer control 14
0xf4	EP14 out buffer control	Spare
0xf8	EP15 in buffer control	Interrupt endpoint buffer control 15
0xfc	EP15 out buffer control	Spare
0x100	EP0 buffer 0 (shared between in and out)	EPx control
0x140	Optional EP0 buffer 1	Spare
0x180	Data buffers	

4.1.2.7.3. Endpoint control register

The endpoint control register is used to configure an endpoint. It contains:

- The endpoint type
- The base address of its data buffer, or data buffers if double buffered
- Interrupts events on the endpoint should trigger

A device must support Endpoint 0 so that it can reply to SETUP packets and be enumerated. As a result, there is no endpoint control register for EP0. Its buffers begin at [0x100](#). All other endpoints can have either single or dual buffers and are mapped at the base address programmed. As EP0 has no endpoint control register, the interrupt enable controls for EP0 come from [SIE_CTRL](#).

Table 395. Endpoint control register layout

Bit(s)	Device Function	Host Function
31	Endpoint Enable	
30	Single buffered (64 bytes) = 0, Double buffered (64 bytes x 2) = 1	
29	Enable Interrupt for every transferred buffer	
28	Enable Interrupt for every 2 transferred buffers (valid for double buffered only)	
27:26	Endpoint Type: Control = 0, ISO = 1, Bulk = 2, Interrupt = 3	
25:18	N/A	The interval the host controller should poll this endpoint. Only applicable for interrupt endpoints. Specified in ms - 1. For example: a value of 9 would poll the endpoint every 10ms.
17	Interrupt on Stall	
16	Interrupt on NAK	
15:6	Address base offset in DPSRAM of data buffer(s)	

NOTE

The data buffer base address must be 64-byte aligned as bits 0-5 are ignored

4.1.2.7.4. Buffer control register

The buffer control register contains information about the state of the data buffers for that endpoint. It is shared between the processor and the controller. If the endpoint is configured to be single buffered, only the first half (bits 0-15) of the buffer are used.

偏移量	设备功能	主机功能
0xe8	EP13 输入缓冲区控制	中断端点缓冲区控制 13
0xec	EP13 输出缓冲区控制	备用
0xf0	缓冲区控制中的 EP14	中断端点缓冲区控制 14
0xf4	EP14 出缓冲区控制	备用
0xf8	缓冲区控制中的 EP15	中断端点缓冲区控制 15
0xfc	EP15 出缓冲区控制	备用
0x100	EP0 缓冲区 0 (输入和输出共用)	端点控制
0x140	可选的 EP0 缓冲区 1	备用
0x180	数据缓冲区	

4.1.2.7.3. 端点控制寄存器

端点控制寄存器用于配置端点，内容包括：

- 端点类型
- 其数据缓冲区的基地址，或如为双缓冲则为数据缓冲区
- 端点应触发的中断事件

设备必须支持端点0，以响应SETUP数据包并完成枚举。因此，EP0没有端点控制寄存器。其缓冲区起始地址为 **0x100**。所有其他端点均可配置为单缓冲或双缓冲，并映射到已编程地址。由于EP0无端点控制寄存器，EP0的中断使能控制来自SIE_CTRL。

表395。端点
控制寄存器布局

位域	设备功能	主机功能
31	端点使能	
30	单缓冲 (64字节) = 0, 双缓冲 (64字节 x 2) = 1	
29	每传输一个缓冲区即使能中断	
28	每传输两个缓冲区 (仅适用于双缓冲) 即使能中断	
27:26	端点类型：控制 = 0, 等时传输 = 1, 批量传输 = 2, 中断传输 = 3	
25:18	不适用	主机控制器轮询此端点的周期。仅适用于中断端点，以毫秒为单位减1指定。例如：值9表示每10毫秒轮询一次此端点。
17	因停滞引发的中断	
16	因NAK响应引发的中断	
15:6	DPSRAM中数据缓冲区的地址基准偏移	

i 注意

数据缓冲区地址必须按64字节对齐，因位0到5被忽略

4.1.2.7.4. 缓冲区控制寄存器

缓冲区控制寄存器包含该端点数据缓冲区状态的信息。该寄存器由处理器与控制器共享。若端点配置为单缓冲，则仅使用缓冲区前半部分（位0至15）。

If double buffering, the buffer select starts at buffer 0. From then on, the buffer select flips between buffer 0 and 1 unless the "reset buffer select" bit is set (which resets the buffer select to buffer 0). The value of the buffer select is internal to the controller and not accessible by the processor.

For host interrupt and isochronous packets on EPx, the buffer full bit will be set on completion even if the transfer was unsuccessful. The error bits in the [SIE_STATUS](#) register can be read to determine the error.

Table 396. Buffer control register layout

Bit(s)	Function
31	Buffer 1 full. Should be set to 1 by the processor for an IN transaction and 0 for an OUT transaction. The controller sets this to 1 for an OUT transaction because it has filled the buffer. The controller sets it to 0 for an IN transaction because it has emptied the buffer. Only valid for double buffered
30	Last buffer of transfer for buffer 1 - only valid for double buffered
29	Data PID for buffer 1 - DATA0 = 0, DATA1 = 1 - only valid for double buffered
27:28	Double buffer offset for Isochronous mode (0 = 128, 1 = 256, 2 = 512, 3 = 1024)
26	Buffer 1 available. Whether the buffer can be used by the controller for a transfer. The processor sets this to 1 when the buffer is configured. The controller sets to 0 when it has used the buffer. i.e. has sent the data to the host for an IN transaction or has filled the buffer with data from the host for an OUT transaction. Only valid for double buffered.
25:16	Buffer 1 transfer length - only valid for double buffered
15	Buffer 0 full. Should be set to 1 by the processor for an IN transaction and 0 for an OUT transaction. The controller sets this to 1 for an OUT transaction because it has filled the buffer. The controller sets it to 0 for an IN transaction because it has emptied the buffer.
14	Last buffer of transfer for buffer 0
13	Data PID for buffer 0 - DATA0 = 0, DATA1 = 1
12	Reset buffer select to buffer 0 - cleared at end of transfer. For DEVICE ONLY
11	Send STALL for device, STALL received for host
10	Buffer 0 available. Whether the buffer can be used by the controller for a transfer. The processor sets this to 1 when the buffer is configured. The controller sets to 0 when it has used the buffer. i.e. has sent the data to the host for an IN transaction or has filled the buffer with data from the host for an OUT transaction.
9:0	Buffer 0 transfer length

⚠️ WARNING

If running `clk_sys` and `clk_usb` at different speeds, the available and stall bits should be set after the other data in the buffer control register. Otherwise the controller may initiate a transaction with data from a previous packet. That is to say, the controller could see the available bit set but get the data pid or length from the previous packet.

4.1.2.8. Device Controller

This section details how the device controller operates when it receives various packet types from the host.

4.1.2.8.1. SETUP

The device controller **MUST** always accept a setup packet from the host. That is why the first 8 bytes of the DPSRAM has dedicated space for the setup packet.

若为双缓冲，缓冲区选择从缓冲区0开始。此后，缓冲区选择在缓冲区0和1间切换，除非“重置缓冲区选择”位被置位（此操作会将缓冲区选择重置为缓冲区0）。缓冲区选择的值为控制器内部状态，处理器无法访问。

对于主机中断及EPx上的等时包，即便传输未成功完成，缓冲区满位仍会在完成时被置位。可读取 SIE_STATUS 寄存器中的错误位以确定错误。

表 396。缓冲区控制寄存器布局

位域	功能
31	缓冲区 1 已满。处理器应将其设置为 1，表示 IN 传输；设置为 0，表示 OUT 传输。控制器将此位设置为 1，表示 OUT 传输，因为缓冲区已填满。 控制器将此位设置为 0，表示 IN 传输，因为缓冲区已清空。仅适用于双缓冲模式
30	缓冲区 1 的传输最后缓冲区标志——仅适用于双缓冲模式
29	缓冲区 1 的数据 PID——DATA0 = 0, DATA1 = 1——仅适用于双缓冲模式
27:28	同步模式的双缓冲区偏移（0 = 128, 1 = 256, 2 = 512, 3 = 1024）
26	缓冲区 1 可用。指示缓冲区是否可被控制器用于传输。缓冲区配置完成后，处理器将其设置为 1。控制器在使用完缓冲区后将其设置为 0，即已将数据发送至主机进行 IN 传输，或已将主机数据填充进缓冲区进行 OUT 传输。仅适用于双缓冲模式。
25:16	缓冲区1传输长度——仅适用于双缓冲模式
15	缓冲区0已满。处理器应将其设置为 1，表示 IN 传输；设置为 0，表示 OUT 传输。控制器将此位设置为 1，表示 OUT 传输，因为缓冲区已填满。 控制器在 IN 传输时将其设置为 0，因为缓冲区已被清空。
14	缓冲区0的最后一个传输缓冲区
13	缓冲区0的数据PID——DATA0 = 0, DATA1 = 1
12	将缓冲区选择重置为缓冲区0——在传输结束时清除。仅适用于设备。
11	设备发送STALL，主机接收STALL。
10	缓冲区0可用。指示缓冲区是否可被控制器用于传输。缓冲区配置完成后，处理器将其设置为 1。控制器在使用完缓冲区后将其设置为 0，即已将数据发送至主机进行 IN 传输，或已将主机数据填充进缓冲区进行 OUT 传输。
9:0	缓冲区0传输长度

● 警告

如果 `clk_sys` 和 `clk_usb` 以不同速度运行，缓冲区控制寄存器中的 `available` 和 `stall` 位应在其它数据之后设置。否则，控制器可能会启动包含前一数据包数据的事务。也就是说，控制器可能会看到 `available` 位被设置，但数据 `pid` 或 `length` 却来自前一数据包。

4.1.2.8. 设备控制器

本节详述设备控制器在接收来自主机的各类数据包时的工作原理。

4.1.2.8.1. SETUP

设备控制器必须始终接受来自主机的 `setup` 数据包。因此，DPSRAM 的前 8 字节为 `setup` 数据包专门预留空间。

The [2] states that receiving a setup packet also clears any stall bits on EP0. For this reason, the stall bits for EP0 are gated with two bits in the [EP_STALL_ARM](#) register. These bits are cleared when a setup packet is received. This means that to send a stall on EP0, you have to set both the stall bit in the buffer control register, and the appropriate bit in [EP_STALL_ARM](#).

Barring any errors, the setup packet will be put into the setup packet buffer at DPSRAM offset [0x0](#). The device controller will then reply with an ACK.

Finally, [SIE_STATUS.SETUP_REC](#) is set to indicate that a setup packet has been received. This will trigger an interrupt if the programmer has enabled the SETUP_REC interrupt (see [INTE](#)).

4.1.2.8.2. IN

From the device's point of view, an [IN](#) transfer means transferring data **INTO** the host. When an [IN](#) token is received from the host the request is handled as follows:

TOKEN phase:

- If [STALL](#) is set in the buffer control register (and if EP0, the appropriate [EP_STALL_ARM](#) bit is set) then send a [STALL](#) response and go back to idle.
- If [AVAILABLE](#) and [FULL](#) bits are set in buffer control move to the phase
- Otherwise send [NAK](#) unless this is an Isochronous endpoint, in which case go to idle.

DATA phase:

- Send DATA. If Isochronous go to idle. Otherwise move to ACK phase.

ACK phase:

- Wait for [ACK](#) packet from host. If there is a timeout then raise a timeout error. If [ACK](#) is received then the packet is done, so move to status phase.

STATUS phase:

- If this was the last buffer in the transfer (i.e. if the [LAST_BUFFER](#) bit in the buffer control register was set), set [SIE_STATUS.TRANS_COMPLETE](#).
- If the endpoint is double buffered, flip the buffer select to the other buffer.
- Set a bit in [BUFF_STATUS](#) to indicate the buffer is done. When handling this event, the programmer should read [BUFF_CPU_SHOULD_HANDLE](#) to see if it is buffer 0 or buffer 1 that is finished. If the endpoint is double buffered it is possible to have both buffers done. The cleared [BUFF_STATUS](#) bit will be set again, and [BUFF_CPU_SHOULD_HANDLE](#) will change in this instance.
- Update status in the appropriate half of the buffer control register: length, pid, and last_buff are set. Everything else is written to zero.

If a [NAK](#) gets sent to the host the host will retry again later.

4.1.2.8.3. OUT

When an [OUT](#) token is received from the host, the request is handled as follows:

TOKEN phase:

- Is the DATA pid what is specified in the buffer control register? If not raise [SIE_STATUS.DATA_SEQ_ERROR](#). (The data pid for an Isochronous endpoint is not checked because Isochronous data is always sent with a [DATA0](#) pid.)
- Is the [AVAILABLE](#) bit set and the [FULL](#) bit unset. If so go to the data phase, unless the [STALL](#) bit is set in which case the device controller will reply with a [STALL](#).

DATA phase:

文献[2]指出，接收setup数据包同时会清除EP0上的任何stall位。因此，EP0的stall位由EP_STALL_ARM寄存器中的两个位进行门控。接收到setup数据包时，这些位将被清除。这意味着，若要在EP0上发送stall信号，必须同时设置缓冲区控制寄存器中的stall位及EP_STALL_ARM中的相应位。

在无错误的情况下，setup包将被存放于DPSRAM偏移 `0x0` 处的setup包缓冲区。设备控制器随后将发送ACK响应。

最后，将设置SIE_STATUS.SETUP_REC，以指示已接收到setup包。若程序员启用了SETUP_REC中断（详见INTE），则此操作将触发中断。

4.1.2.8.2. IN

从设备角度来看，`IN`传输指的是将数据传输进入主机。当从主机接收到`IN`令牌时，请求将按以下方式处理：

TOKEN阶段：

- 若缓冲区控制寄存器中设置了`STALL`（且对于EP0，相应的EP_STALL_ARM位已设置），则发送STALL响应并返回空闲状态。
- 如果缓冲区控制中设置了`AVAILABLE`和`FULL`位，则进入该阶段。
- 否则发送`NAK`，除非该端点为等时端点，在此情况下进入空闲状态。

数据阶段：

- 发送数据。若为等时传输，则进入空闲状态。否则进入确认(ACK)阶段。

确认(ACK)阶段：

- 等待主机发送的`ACK`包。若超时，则产生超时错误。接收到`ACK`后，数据包传输完成，转入状态阶段。

状态阶段：

- 如果这是本次传输的最后一个缓冲区（即缓冲区控制寄存器中的`LAST_BUFFER`位被设置），则设置SIE_STATUS.TRANS_COMPLETE。
- 若端点为双缓冲，则切换缓冲区选择至另一缓冲区。
- 在BUFF_STATUS中设置相应位以指示缓冲区已完成。处理该事件时，程序员应读取BUFF_CPU_SHOULD_HANDLE以判断完成的是缓冲区0还是缓冲区1。如果端点采用双缓冲，两个缓冲区均可能已完成。已清除的BUFF_STATUS位将再次被置位，并且此时BUFF_CPU_SHOULD_HANDLE将发生变化。**
- 更新缓冲区控制寄存器对应半部分的状态：`length`、`pid` 及 `last_buff` 被设置，其余字段写为零。

若向主机发送了`NAK`，主机将稍后重新尝试。

4.1.2.8.3. OUT

当从主机接收到`OUT`令牌时，按如下方式处理请求：

TOKEN阶段：

- `DATA pid` 是否与缓冲区控制寄存器中指定的值一致？若不一致，则触发SIE_STATUS.DATA_SEQ_ERROR。（Isochronous端点的数据`pid`不进行检查，因为Isochronous数据始终以`DATA0 pid`发送。）
- `AVAILABLE`位是否被设置且`FULL`位是否未设置？若是，则进入数据阶段；除非`STALL`位被置位，此时设备控制器将以`STALL`响应。

数据阶段：

- Store received data in buffer. If Isochronous go to STATUS phase. Otherwise go to ACK phase.

ACK phase:

- Send ACK. Go to STATUS phase.

STATUS phase:

See status phase from [Section 4.1.2.8.2](#). The only difference is that the **FULL** bit is set in the buffer control register to indicate that data has been received whereas in the **IN** case the **FULL** bit is cleared to indicate that data has been sent.

4.1.2.8.4. Suspend and Resume

The USB device controller supports both suspend and resume, as well as remote resume (triggered with **SIE_CTRL.RESUME**), where the device initiates the resume. There is an interrupt / status bit in **SIE_STATUS**. It is not necessary to enable the suspend and resume interrupts, as most devices do not need to care about suspend and resume.

The device goes into suspend when it does not see any start of frame packets (transmitted every 1ms) from the host.

NOTE

If you enable the suspend interrupt, it is likely you will see a suspend interrupt when the device is first connected but the bus is idle. The bus can be idle for a few ms before the host begins sending start of frame packets. You will also see a suspend interrupt when the device is disconnected if you do not have a VBUS detect circuit connected. This is because without VBUS detection, it is impossible to tell the difference between being disconnected and suspended.

4.1.2.8.5. Errata

There are two hardware issues with the device controller, both of which have software workarounds on RP2040B0, RP2040B1, and are fixed in hardware on RP2040B2. See [RP2040-E2](#) and [RP2040-E5](#) for more information.

4.1.2.9. Host Controller

The host controller design is similar to the device controller. All transactions are started by the host, so the host is always dealing with transactions it has started. For this reason there is only one set of endpoint control / endpoint buffer control registers. There is also additional hardware to poll interrupt endpoints in the background when there are no software controlled transactions taking place.

The host needs to send keep-alive packets to the device every 1ms to keep the device from suspending. In Full Speed mode this is done by sending a **SOF** (start of frame) packet. In Low Speed mode, an **EOP** (end of packet) is sent. When setting up the controller, **SIE_CTRL.KEEP_ALIVE_EN** and **SIE_CTRL.SOF_EN** should be set to enable these packets.

Several bits in **SIE_CTRL** are used to begin a host transaction:

- **SEND_SETUP** - Send a setup packet. This is typically used in conjunction with **RECEIVE_TRANS** so the setup packet will be sent followed by the additional data transaction expected from the device.
- **SEND_TRANS** - This transfer is **OUT** from the host
- **RECEIVE_TRANS** - This transfer is **IN** to the host
- **START_TRANS** - Start the transfer - non-latching
- **STOP_TRANS** - Stop the current transfer - non-latching
- **PREAMBLE_ENABLE** - Use this to send a packet to a Low Speed device on a Full Speed hub. This will send a **PRE** token packet before every packet the host sends (i.e. pre, token, pre, data, pre, ack).
- **SOF_SYNC** - The SOF Sync bit is used to delay the transaction until after the next SOF. This is useful for interrupt and isochronous endpoints. The Host controller prevents a transaction of 64bytes from clashing with the SOF packets.

- 将接收的数据存储到缓冲区。若为等时传输，则进入状态阶段。否则进入ACK阶段。

确认(ACK)阶段：

- 发送ACK，进入状态阶段。

状态阶段：

详见第4.1.2.8.2节的状态阶段。唯一区别在于缓冲区控制寄存器中的 FULL位被设置，表示数据已接收，而在 IN情况下， FU LL位则被清除，表示数据已发送。

4.1.2.8.4. 挂起与恢复

USB设备控制器支持挂起及恢复功能，并支持远程恢复（由SIE_CTRL.RESUME触发），设备可以主动发起恢复。SIE_STATUS寄存器含有一个中断/状态位。通常无需启用挂起和恢复中断，因为大多数设备无需处理挂起与恢复。

当设备未检测到主机每1毫秒发送的帧起始包时，设备进入挂起状态。

i 注意

如果启用挂起中断，则设备首次连接且总线处于空闲状态时，您很可能会收到挂起中断。在主机开始发送帧起始数据包之前，总线可能会空闲数毫秒。如果未连接VBUS检测电路，设备断开时也会产生挂起中断。这是因为缺乏VBUS检测时，无法区分设备断开与挂起状态。

4.1.2.8.5. 勘误

设备控制器存在两个硬件问题，RP2040B0和RP2040B1通过软件进行规避，RP2040B2已在硬件层面修复。详情请参阅RP2040-E2和RP2040-E5。

4.1.2.9. 主机控制器

主机控制器的设计与设备控制器类似。所有事务均由主机发起，主机始终处理其自身发起的事务。因此，只有一组端点控制／端点缓冲区控制寄存器。还有额外硬件用于在无软件控制事务进行时在后台轮询中断端点。

主机需每1毫秒向设备发送保持活动包，以防止设备进入挂起状态。在全速模式下，通过发送一个 SOF（帧开始）包实现此操作。在低速模式下，发送一个 EOP（数据包结束）信号。配置控制器时，应启用SIE_CTRL.KEEP_ALIVE_EN和SIE_CTRL.SOF_EN位以允许这些数据包。

SIE_CTRL寄存器中的若干位用于启动主机事务：

- SEND_SETUP- 发送一个setup包。该命令通常与RECEIVE_TRANS结合使用，先发送setup包，随后进行设备预期的附加数据传输。
- SEND_TRANS- 该传输方向为来自主机的 OUT
- RECEIVE_TRANS- 该传输方向为至主机的 IN
- START_TRANS - 启动传输 - 非锁存
- STOP_TRANS - 停止当前传输 - 非锁存
- PREAMBLE_ENABLE- 用于向全速集线器上的低速设备发送数据包。该设置会在主机发送的每个数据包之前发送一个 PRE E token数据包（即 pre, token, pre, data, pre, ack）。
- SOF_SYNC- SOF同步位用于延迟事务，直至下一个SOF发生后。这对于中断和同步传输端点尤为有用。主机控制器防止64字节事务与SOF数据包发生冲突。

For longer Isochronous packet the software is responsible for preventing a collision by using the SOF Sync bit and limiting the number of packets sent in one frame. If a transaction is set up with multiple packets the SOF Sync bit only applies to the first packet.

⚠️ WARNING

The `START_TRANS` bit is synchronised separately to other control bits in the `SIE_CTRL` register. The `START_TRANS` bit should be set separately to the rest of the data in the `SIE_CTRL` register, so that the register contents are stable when the controller is prompted to start a transfer. This is necessary because the processor clock `clk_sys` can be asynchronous to the `clk_usb` clock.

- Write fields in `SIE_CTRL` apart from `START_TRANS`
- `nop` for some `clk_sys` cycles to ensure that at least two `clk_usb` cycles have passed. For example if `clk_sys` was running at 125MHz and `clk_usb` was running at 48MHz then 125/48 rounded up would be 6 `nop` instructions
- Set the `START_TRANS` bit.

4.1.2.9.1. SETUP

The `SETUP` packet sent from the host always comes from the dedicated 8 bytes of space at offset `0x0` of the DPSRAM. Like the device controller, there are no control registers associated with the setup packet. The parameters are hard coded and loaded into the hardware when you write to `START_TRANS` with the `SEND_SETUP` bit set. Once the setup packet has been sent, the host state machine will wait for an `ACK` from the device. If there is a timeout then an `RX_TIMEOUT` error will be raised. If the `SEND_TRANS` bit is set then the host state machine will move to the `OUT` phase. Most commonly the `SEND_SETUP` packet is used in conjunction with the `RECEIVE_TRANS` bit and will therefore move to the `IN` phase after sending a setup packet.

4.1.2.9.2. IN

An `IN` transfer is triggered with the `RECEIVE_TRANS` bit set when the `START_TRANS` bit is set. This may be preceded by a `SETUP` packet being sent if the `SEND_SETUP` bit was set.

CONTROL phase:

- Read `EPx control` register located at `0x80` to get the endpoint information:
 - Are we double buffered?
 - What interrupts to enable
 - Base address of the data buffer, or data buffers if in double buffered mode
 - Endpoint type
- Read `EPx buffer control` register at `0x100` to get the endpoint buffer information such as transfer length and data pid. The host state machine still checks for the presence of the `AVAILABLE` bit, so this needs to be set and `FULL` needs to be unset. The transaction will not happen until this is the case.

TOKEN phase:

- Send the `IN` token packet to the device. The target device address and endpoint come from the `ADDR_ENDP` register.

DATA phase:

- Receive the first data packet from the device. Raise RX timeout error if the device doesn't reply. Raise DATA SEQ ERROR if the data packet has wrong DATA PID.

ACK phase:

- Send ACK to device

对于较长的同步数据包，软件需负责使用SOF同步位并限制单帧内发送数据包的数量，以防止冲突。若事务中包含多个数据包，SOF同步位仅适用于第一个数据包。

● 警告

START_TRANS位与SIE_CTRL寄存器中的其他控制位分别同步。应将SIE_CTRL寄存器中的START_TRANS位与其他数据分开设置，以确保在控制器被触发开始传输时，寄存器内容保持稳定。这是必要的，因为处理器时钟clk_sys可能与clk_usb时钟异步。

- 写入 SIE_CTRL 中除 START_TRANS 以外的字段
- 执行nop操作，持续若干clk_sys周期，以确保至少经过两个clk_usb周期。例如，如果clk_sys运行于125MHz，而clk_usb运行于48MHz，则125/48向上取整为6，即6条nop指令。
- 设置START_TRANS位。

4.1.2.9.1. SETUP

从主机发送的 SETUP 数据包始终来自DPSRAM中偏移地址 0x0 处专用的8字节空间。

与设备控制器类似，没有与 SETUP 数据包相关联的控制寄存器。这些参数为硬编码形式，并在您写入且设置了 SEND_SETUP 位的 START_TRANS 时加载到硬件中。一旦设置包发送完成，主机状态机将等待设备返回 ACK。若发生超时，则会触发 RX_TIMEOUT 错误。若设置了 SEND_TRANS 位，主机状态机将进入 OUT 阶段。通常，SEND_SETUP 包与 RECEIVE_TRANS 位配合使用，因此发送设置包后将进入 IN 阶段。

4.1.2.9.2. IN

当设置了 START_TRANS 位与 RECEIVE_TRANS 位时，将触发一次 IN 传输。若先前设置了 SEND_SETUP 位，可能会发送一个 SETUP 包。

控制阶段：

- 读取位于 0x80 的 EPx control 寄存器以获取端点信息：
 - 是否启用双缓冲？
 - 需启用的中断
 - 数据缓冲区基地址，或双缓冲模式下的数据缓冲区
 - 端点类型
- 读取位于 0x100 的 EPx 缓冲区控制 寄存器以获取端点缓冲区信息，例如传输长度和数据 PID。主机状态机仍会检查 AVAILABLE 位的存在，因此该位须设为 1，且 FULL 位须清零。事务仅在满足此条件时发生。

TOKEN阶段：

- 向设备发送 IN 令牌包。目标设备地址和端点取自 ADDR_ENDP 寄存器。

数据阶段：

- 接收来自设备的首个数据包。若设备无响应，触发 RX 超时错误。如数据包的数据 PID 错误，触发数据序列错误。

确认(ACK)阶段：

- 向设备发送 ACK 信号

STATUS phase:

- Set **BUFF_STATUS** bit and update buffer control register. Will set **FULL**, **LAST_BUFF** if applicable, **DATA_PID**, **WR_LEN**. **TRANS_COMPLETE** will be set if this is the last buffer in the transfer.

CONTROL phase (pt 2):

- The host state machine will keep performing **IN** transactions until **LAST_BUFF** is seen in the buffer_control register. If the host is in double buffered mode then the host controller will toggle between **BUF0** and **BUF1** sections of the buffer control register. Otherwise it will keep reading the buffer control register for buffer 0 and wait for the **FULL** to be unset and **AVAILABLE** to be set before starting the next **IN** transaction (i.e. wait in the control phase). The device can send a zero length packet to the host to indicate that it has no more data. In which case the host state machine will stop listening for more data regardless of if the **LAST_BUFF** flag was set or not. The host software can tell this has happened because **BUFF_DONE** will be set with a data length of 0 in the buffer control register.

– WARNING

The USB host controller has a bug ([RP2040-E4](#)) that means the status written back to the buffer control register can appear in the wrong half of the register. Bits 0-15 are for buffer 0, and bits 16-31 are for buffer 1. The host controller has a buffer selector that is flipped after each transfer is complete. This buffer selector is incorrectly used when writing status information back to the buffer control register even in single buffered mode. The buffer selector is not used when reading the buffer control register. The implication of this is that host software needs to keep track of the buffer selector and shift the buffer control register to the right by 16 bits if the buffer selector is 1.

For more information, see [RP2040-E4](#).

4.1.2.9.3. OUT

An **OUT** transfer is triggered with the **SEND_TRANS** bit set when the **START_TRANS** bit is set. This may be preceded by a **SETUP** packet being sent if the **SEND_SETUP** bit was set.

CONTROL phase:

- Read **EPx control** to get endpoint information (same as [Section 4.1.2.9.2](#))
- Read **EPx buffer control** to get the transfer length, data pid. **AVAILABLE** and **FULL** must be set for the transfer to start.

TOKEN phase

- Send **OUT** packet to the device. The target device address and endpoint come from the **ADDR_ENDP** register.

DATA phase:

- Send the first data packet to the device. If the endpoint type is Isochronous then there is no ACK phase so the host controller will go straight to status phase. If **ACK** received then go to status phase. Otherwise:
 - If no reply is received then raise **SIE_STATUS.RX_TIMEOUT**.
 - If **NAK** received raise **SIE_STATUS.NAK_REC** and send the data packet again.
 - If **STALL** received then raise **SIE_STATUS.STALL_REC** and go to idle.

STATUS phase:

- Set **BUFF_STATUS** bit and update buffer control register. **FULL** will be set to 0. **TRANS_COMPLETE** will be set if this is the last buffer in the transfer.

状态阶段：

- 设置BUFF_STATUS位并更新缓冲区控制寄存器。将设置 FULL、 LAST_BUFF（如适用）、 DATA_PID及 WR_LEN。若为传输中的最后一个缓冲区，将设置TRANS_COMPLETE。

控制阶段（第2部分）：

- 主机状态机将持续执行 IN事务，直到在buffer_control寄存器中检测到 LAST_BUFF。若主机处于双缓冲模式，主机控制器将在buffer_control寄存器的 BUFO和 BUF1部分之间切换。否则，将持续读取缓冲区控制寄存器的缓冲区0，等待 FULL标志清除且 AVAILABLE标志设置后，再启动下一次IN事务（即在控制阶段等待）。设备可向主机发送零长度数据包，以示其无更多数据。此时，无论 LAST_BUFF标志是否设置，主机状态机均停止接收更多数据。主机软件可通过检测缓冲区控制寄存器中数据长度为0且 BUFF_DONE标志被设置来确认此情况。

● 警告

USB主机控制器存在一个漏洞（RP2040-E4），导致写回缓冲区控制寄存器的状态可能出现在寄存器的错误半区。位0至15用于缓冲区0，位16至31用于缓冲区1。主机控制器设有缓冲区选择器，每次传输完成后会切换该选择器。即使在单缓冲模式下，写回状态信息至缓冲区控制寄存器时，该缓冲区选择器仍被错误使用。读取缓冲区控制寄存器时不使用缓冲区选择器。由此可见，主机软件需跟踪缓冲区选择器状态，若缓冲区选择器为1，则应将缓冲区控制寄存器右移16位。

详细信息请参阅 RP2040-E4。

4.1.2.9.3. OUT

当 START_TRANS位置位且 SEND_TRANS位被设置时，将触发一个OUT传输。若 SEND_SETUP位已设置，则之前可能会发送一个 SETUP包。

控制阶段：

- 读取EPx control以获取端点信息（同4.1.2.9.2节）。
- 读取EPx缓冲区控制以获取传输长度和数据pid。 AVAILABLE和 FULL必须被设置，传输方可开始。

TOKEN阶段

- 向设备发送 OUT数据包。目标设备地址和端点取自ADDR_ENDP寄存器。

数据阶段：

- 向设备发送首个数据包。若端点类型为等时传输，则无ACK阶段，主控制器将直接进入状态阶段。若收到 ACK，则进入状态阶段。否则：
 - 若未收到回复，则触发SIE_STATUS.RX_TIMEOUT。
 - 若收到 NAK，则触发SIE_STATUS.NAK_REC并重新发送数据包。
 - 若收到 STALL，则触发SIE_STATUS.STALL_REC并进入空闲状态。

状态阶段：

- 设置BUFF_STATUS位并更新缓冲区控制寄存器， FULL位将被置为0。若为传输中的最后一个缓冲区，将设置TRANS_COMPLETE。

⚠️ WARNING

The bug mentioned above ([RP2040-E4](#)) in the IN section also applies to the OUT section.

CONTROL phase (pt 2):

If this isn't the last buffer in the transfer then wait for **FULL** and **AVAILABLE** to be set in the *EPx buffer control* register again.

4.1.2.9.4. Interrupt Endpoints

The host controller can poll interrupt endpoints on many devices (up to a maximum of 15 endpoints). To enable these, the programmer must:

- Pick the next free interrupt endpoint slot on the host controller (starting at 1, to a maximum of 15)
- Program the appropriate endpoint control register and buffer control register like you would with a normal **IN** or **OUT** transfer. Note that interrupt endpoints are only single buffered so the **BUF1** part of the buffer control register is invalid.
- Set the address and endpoint of the device in the appropriate **ADDR_ENDP** register (**ADDR_ENDP1** to **ADDR_ENDP15**). The preamble bit should be set if the device is Low Speed but attached to a Full Speed hub. The endpoint direction bit should also be set.
- Set the interrupt endpoint active bit in **INT_EP_CTRL** (i.e. set bit 1 to 15 of that register)

Typically an interrupt endpoint will be an **IN** transfer. For example, a USB hub would be polled to see if the state of any of its ports have changed. If there is no change the hub will reply with a **NAK** to the controller and nothing will happen. Similarly, a mouse will reply with a **NAK** unless the mouse has been moved since the last time the interrupt endpoint was polled.

Interrupt endpoints are polled by the controller once a **SOF** packet has been sent by the host controller.

The controller loops from 1 to 15 and will attempt to poll any interrupt endpoint with the **EP_ACTIVE** bit set to 1 in **INT_EP_CTRL**. The controller will then read the endpoint control register, and buffer control register to see if there is an available buffer (i.e. **FULL + AVAILABLE** if an **OUT** transfer and **NOT FULL + AVAILABLE** for an **IN** transfer). If not, the controller will move onto the next interrupt endpoint slot.

If there is an available buffer, then the transfer is dealt with the same as a normal **IN** or **OUT** transfer and the **BUFF_DONE** flag in **BUFF_STATUS** will be set when the interrupt endpoint has a valid buffer. **BUFF_CPU_SHOULD_HANDLE** is invalid for interrupt endpoints as there is only a single buffer that can ever be done ([RP2040-E3](#)).

4.1.2.10. VBUS Control

The USB controller can be connected up to GPIO pins (see [Section 2.19](#)) for the purpose of VBUS control:

- VBUS enable, used to enable VBUS in host mode. VBUS enable is set in **SIE_CTRL**
- VBUS detect, used to detect that VBUS is present in device mode. VBUS detect is a bit in **SIE_STATUS** and can also raise a **VBUS_DETECT** interrupt (enabled in **INTE**)
- VBUS overcurrent, used to detect an overcurrent event. Applicable to both device and host. VBUS overcurrent is a bit in **SIE_STATUS**.

It is not necessary to connect up any of these pins to GPIO. The host can permanently supply VBUS and detect a device being connected when either the DP or DM pin is pulled high. VBUS detect can be forced in **USB_PWR**.

4.1.3. Programmer's Model

● 警告

上述IN部分所述的错误（RP2040-E4）同样适用于OUT部分。

控制阶段（第2部分）：

若此次传输尚未结束，则等待 FULL和 AVAILABLE在EPx缓冲区控制寄存器中再次被设置。

4.1.2.9.4. 中断端点

主机控制器可以轮询多个设备上的中断端点（最大为15个端点）。要启用这些功能，程序员必须：

- 在主机控制器上选择下一个空闲的中断端点槽（从1开始，最多15个）
- 如同普通的 IN或 OUT传输一般，配置相应的端点控制寄存器和缓冲区控制寄存器。注意，中断端点仅支持单缓冲，因此缓冲区控制寄存器中的BUF1部分无效。
- 在对应的 ADDR_ENDP 寄存器（ADDR_ENDP1至ADDR_ENDP15）中设置设备地址及端点号。
若设备为低速且接入全速集线器，需设置前导码位并同时设置端点方向位。
- 在INT_EP_CTRL寄存器中设置中断端点激活位（即寄存器的第1至第15位）。

通常，中断端点为 IN方向传输。例如，USB集线器会被轮询以检查其任一端口的状态是否发生变化。如无变化，集线器将向控制器回复 NAK，且无任何操作发生。

同理，除非鼠标自上次轮询中断端点以来移动过，否则鼠标将回复 NAK。

中断端点由控制器在主机控制器发送完一个 SOF数据包后进行轮询。

控制器循环遍历1至15，尝试轮询在INT_EP_CTRL寄存器中EP_ACTIVE位为1的所有中断端点。随后，控制器读取端点控制寄存器及缓冲控制寄存器，以确认是否存在可用缓冲区（即，对于OUT传输，应为FULL且AVAILABLE；对于IN传输，应为 NOT FULL且AVAILABLE）。如无可用缓冲区，控制器将继续轮询下一个中断端点槽。

若存在可用缓冲区，传输将按正常的 IN或 OUT传输处理，当中断端点拥有有效缓冲区时， BUFF_DONE标志将在BUFF_STATUS寄存器中被置位。BUF_CPU_SHOULD_HANDLE 对中断端点无效，因为仅能完成单个缓冲区（RP2040-E3）。

4.1.2.10. VBUS 控制

USB控制器可连接至 GPIO 引脚（详见第 2.19 节），以实现 VBUS 控制：

- VBUS 使能，用于在主机模式下启用 VBUS。VBUS 使能位于 SIE_CTRL 寄存器中。
- VBUS 检测，用于在设备模式下检测 VBUS 的存在。VBUS 检测为 SIE_STATUS 寄存器中的一位，并可触发 VBUS_DETECT 中断（由 INTE 使能）。
- VBUS 过流，用于检测过流事件。适用于设备和主机两种模式。VBUS 过流为 SIE_STATUS 寄存器中的一位。

无须将上述引脚连接至 GPIO。主机可持续供电 VBUS，并通过 DP 或 DM 引脚电平拉高以检测设备连接。VBUS检测可在 USB_PWR中被强制执行。

4.1.3. 程序员模型

4.1.3.1. TinyUSB

The RP2040 TinyUSB port should be considered as the reference implementation for this USB controller. This port can be found in:

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/dcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/hcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/rp2040_usb.h

4.1.3.2. Standalone device example

A standalone USB device example, `dev_lowlevel`, makes it easier to understand how to interact with the USB controller without needing to understand the TinyUSB abstractions. In addition to endpoint 0, the standalone device has two bulk endpoints: EP1 OUT and EP2 IN. The device is designed to send whatever data it receives on EP1 to EP2. The example comes with a small Python script that writes "Hello World" into EP1 and checks that it is correctly received on EP2.

The code included in this section will walk you through setting up to the USB device controller to receive a setup packet, and then respond to the setup packet.

Figure 58. USB analyser trace of the `dev_lowlevel` USB device example. The control transfers are the device enumeration. The first bulk OUT (out from the host) transfer, highlighted in blue, is the host sending "Hello World" to the device. The second bulk transfer IN (in to the host), is the device returning "Hello World" to the host.

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors
0	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors
1	S	SET	0	0	SET_ADDRESS	New address 7	0x0000	0
2	S	GET	7	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
3	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor
4	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	4 Descriptors
5	S	GET	7	0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	Lang Supported
6	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	Pico Test Device
7	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 1	Language ID 0x0409	Raspberry Pi
8	S	SET	7	0	SET_CONFIGURATION	New Configuration 1	0x0000	0
9	S	Bulk	ADDR	ENDP	Bytes Transferred			
10	S	Bulk	IN	7	12			

4.1.3.2.1. Device controller initialisation

The following code initialises the USB device.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 183 - 217

```

183 void usb_device_init() {
184     // Reset usb controller
185     reset_unreset_block_num_wait_blocking(RESET_USBCTRL);
186
187     // Clear any previous state in dpram just in case
188     memset(usb_dpram, 0, sizeof(*usb_dpram)); ①
189
190     // Enable USB interrupt at processor
191     irq_set_enabled(USBCTRL_IRQ, true);
192
193     // Mux the controller to the onboard usb phy

```

4.1.3.1. TinyUSB

RP2040 TinyUSB端口应视为该USB控制器的参考实现。该端口可在以下位置找到：

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/dcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/hcd_rp2040.c

https://github.com/hathach/tinyusb/blob/master/src/portable/raspberrypi/rp2040/rp2040_usb.h

4.1.3.2. 独立设备示例

独立USB设备示例`dev_lowlevel`，有助于理解如何与USB控制器交互，而无需理解TinyUSB抽象层。除了端点0外，独立设备还具有两个批量传输端点：EP1 OUT和EP2 IN。该设备设计为将从EP1接收的所有数据发送至EP2。示例附带一个简易Python脚本，将“Hello World”写入EP1，并验证该数据是否被正确接收到EP2。本节代码将引导您设置USB设备控制器，以接收设置包并响应该设置包。

图58。dev_lowlevel USB设备示例的USB分析仪跟踪。控制传输即设备枚举。第一个批量OUT（从主机发出）传输，蓝色高亮部分，表示主机向设备发送“Hello World”。第二个批量IN（传入主机）传输，表示设备将“Hello World”返回给主机。

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors
0	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
1	S	SET	0	0	SET_ADDRESS	New address 7	0x0000	
2	S	GET	7	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor
3	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor
4	S	GET	7	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	4 Descriptors
5	S	GET	7	0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	Lang Supported
6	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	Pico Test Device
7	S	GET	7	0	GET_DESCRIPTOR	STRING type, Index 1	Language ID 0x0409	Raspberry Pi
8	S	SET	7	0	SET_CONFIGURATION	New Configuration 1	0x0000	
9	S	Bulk	ADDR	ENDP	Bytes Transferred			
10	S	OUT	7	1		12		
		Bulk	ADDR	ENDP	Bytes Transferred			
		IN	7	2		12		

4.1.3.2.1. 设备控制器初始化

以下代码用于初始化USB设备。

Pico示例代码：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第183至217行

```

183 void usb_device_init() {
184     // 重置USB控制器
185     reset_unreset_block_num_wait_blocking(RESET_USBCTRL);
186
187     // 清除dpram中的任何先前状态
188     memset(usb_dpram, 0, sizeof(*usb_dpram)); ①
189
190     // 在处理器上启用USB中断
191     irq_set_enabled(USBCTRL_IRQ, true);
192
193     // 将控制器复用于板载USB物理层

```

```

194     usb_hw->muxing = USB_USB_MUXING_TO_PHY_BITS | USB_USB_MUXING_SOFTCON_BITS;
195
196     // Force VBUS detect so the device thinks it is plugged into a host
197     usb_hw->pwr = USB_USB_PWR_VBUS_DETECT_BITS | USB_USB_PWR_VBUS_DETECT_OVERRIDE_EN_BITS;
198
199     // Enable the USB controller in device mode.
200     usb_hw->main_ctrl = USB_MAIN_CTRL_CONTROLLER_EN_BITS;
201
202     // Enable an interrupt per EP0 transaction
203     usb_hw->sie_ctrl = USB_SIE_CTRL_EP0_INT_1BUF_BITS; ②
204
205     // Enable interrupts for when a buffer is done, when the bus is reset,
206     // and when a setup packet is received
207     usb_hw->inte = USB_INTS_BUFF_STATUS_BITS |
208                     USB_INTS_BUS_RESET_BITS |
209                     USB_INTS_SETUP_REQ_BITS;
210
211     // Set up endpoints (endpoint control registers)
212     // described by device configuration
213     usb_setup_endpoints();
214
215     // Present full speed device by enabling pull up on DP
216     usb_hw_set->sie_ctrl = USB_SIE_CTRL_PULLUP_EN_BITS;
217 }

```

4.1.3.2.2. Configuring the endpoint control registers for EP1 and EP2

The function `usb_configure_endpoints` loops through each endpoint defined in the device configuration (including EP0 in and EP0 out, which don't have an endpoint control register defined) and calls the `usb_configure_endpoint` function. This sets up the endpoint control register for that endpoint:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 149 - 164

```

149 void usb_setup_endpoint(const struct usb_endpoint_configuration *ep) {
150     printf("Set up endpoint 0x%x with buffer address 0x%p\n", ep->descriptor-
>bEndpointAddress, ep->data_buffer);
151
152     // EP0 doesn't have one so return if that is the case
153     if (!ep->endpoint_control) {
154         return;
155     }
156
157     // Get the data buffer as an offset of the USB controller's DPRAM
158     uint32_t dpram_offset = usb_buffer_offset(ep->data_buffer);
159     uint32_t reg = EP_CTRL_ENABLE_BITS
160                 | EP_CTRL_INTERRUPT_PER_BUFFER
161                 | (ep->descriptor->bmAttributes << EP_CTRL_BUFFER_TYPE_LSB)
162                 | dpram_offset;
163     *ep->endpoint_control = reg;
164 }

```

4.1.3.2.3. Receiving a setup packet

An interrupt is raised when a setup packet is received, so the interrupt handler must handle this event:

```

194     usb_hw->muxing = USB_USB_MUXING_TO_PHY_BITS | USB_USB_MUXING_SOFTCON_BITS;
195
196     // 强制VBUS检测，使设备识别为已连接主机
197     usb_hw->pwr = USB_USB_PWR_VBUS_DETECT_BITS | USB_USB_PWR_VBUS_DETECT_OVERRIDE_EN_BITS;
198
199     // 以设备模式启用USB控制器
200     usb_hw->main_ctrl = USB_MAIN_CTRL_CONTROLLER_EN_BITS;
201
202     // 使能每个EP0事务的中断
203     usb_hw->sie_ctrl = USB_SIE_CTRL_EP0_INT_1BUF_BITS; ②
204
205     // 启用缓冲区完成、总线复位
206     // 以及接收setup包时的中断
207     usb_hw->inte = USB_INTS_BUFF_STATUS_BITS |
208                     USB_INTS_BUS_RESET_BITS |
209                     USB_INTS_SETUP_REQ_BITS;
210
211     // 设置端点（端点控制寄存器）
212     // 由设备配置描述
213     usb_setup_endpoints();
214
215     // 通过在DP线上启用上拉，激活全速设备
216     usb_hw_set->sie_ctrl = USB_SIE_CTRL_PULLUP_EN_BITS;
217 }

```

4.1.3.2.2. 配置EP1和EP2的端点控制寄存器

函数`usb_configure_endpoints`循环遍历设备配置中定义的每个端点（包括未定义端点控制寄存器的EP0输入和EP0输出），并调用`usb_configure_endpoint`函数。此操作设置该端点的端点控制寄存器：

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第149至164行

```

149 void usb_setup_endpoint(const struct usb_endpoint_configuration *ep) {
150     printf("设置端点 0x%02x, 缓冲区地址 0x%p\n", ep->descriptor->bEndpointAddress, ep->data_buffer); 151
151
152     // EP0 不具备此项，如是则返回
153     if (!ep->endpoint_control) {
154         return;
155     }
156
157     // 获取 USB 控制器 DPRAM 中的缓冲区偏移
158     uint32_t dpram_offset = usb_buffer_offset(ep->data_buffer);
159     uint32_t reg = EP_CTRL_ENABLE_BITS
160                 | EP_CTRL_INTERRUPT_PER_BUFFER
161                 | (ep->descriptor->bmAttributes << EP_CTRL_BUFFER_TYPE_LSB)
162                 | dpram_offset;
163     *ep->endpoint_control = reg;
164 }

```

4.1.3.2.3. 接收设置包

接收到设置包时会触发中断，故中断处理程序必须处理此事件：

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 494 - 504

```

494 void isr_usbctrl(void) {
495     // USB interrupt handler
496     uint32_t status = usb_hw->ints;
497     uint32_t handled = 0;
498
499     // Setup packet received
500     if (status & USB_INTS_SETUP_REQ_BITS) {
501         handled |= USB_INTS_SETUP_REQ_BITS;
502         usb_hw_clear->sie_status = USB_SIE_STATUS_SETUP_REC_BITS;
503         usb_handle_setup_packet();
504     }

```

The setup packet gets written to the first 8 bytes of the USB ram, so the setup packet handler casts that area of memory to `struct usb_setup_packet *`.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 383 - 427

```

383 void usb_handle_setup_packet(void) {
384     volatile struct usb_setup_packet *pkt = (volatile struct usb_setup_packet *) &usb_dpram
385     ->setup_packet;
386     uint8_t req_direction = pkt->bmRequestType;
387     uint8_t req = pkt->bRequest;
388
389     // Reset PID to 1 for EP0 IN
390     usb_get_endpoint_configuration(EP0_IN_ADDR)->next_pid = 1u;
391
392     if (req_direction == USB_DIR_OUT) {
393         if (req == USB_REQUEST_SET_ADDRESS) {
394             usb_set_device_address(pkt);
395         } else if (req == USB_REQUEST_SET_CONFIGURATION) {
396             usb_set_device_configuration(pkt);
397         } else {
398             usb_acknowledge_out_request();
399             printf("Other OUT request (0x%x)\r\n", pkt->bRequest);
400         }
401     } else if (req_direction == USB_DIR_IN) {
402         if (req == USB_REQUEST_GET_DESCRIPTOR) {
403             uint16_t descriptor_type = pkt->wValue >> 8;
404
405             switch (descriptor_type) {
406                 case USB_DT_DEVICE:
407                     usb_handle_device_descriptor(pkt);
408                     printf("GET DEVICE DESCRIPTOR\r\n");
409                     break;
410
411                 case USB_DT_CONFIG:
412                     usb_handle_config_descriptor(pkt);
413                     printf("GET CONFIG DESCRIPTOR\r\n");
414                     break;
415
416                 case USB_DT_STRING:
417                     usb_handle_string_descriptor(pkt);
418                     printf("GET STRING DESCRIPTOR\r\n");
419                     break;
420
421                 default:
422                     printf("Unhandled GET_DESCRIPTOR type 0x%x\r\n", descriptor_type);
423             }
424         } else {
425             printf("Other IN request (0x%x)\r\n", pkt->bRequest);

```

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第494至504行

```

494 void isr_usbctrl(void) {
495     // USB 中断处理函数
496     uint32_t status = usb_hw->ints;
497     uint32_t handled = 0;
498
499     // 设置数据包已接收
500     if (status & USB_INTS_SETUP_REQ_BITS) {
501         handled |= USB_INTS_SETUP_REQ_BITS;
502         usb_hw_clear->sie_status = USB_SIE_STATUS_SETUP_REC_BITS;
503         usb_handle_setup_packet();
504     }

```

设置数据包被写入 USB RAM 的前 8 个字节，故设置数据包处理函数将该内存区域转换为 `struct usb_setup_packet *` 类型。

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第 383-427 行

```

383 void usb_handle_setup_packet(void) {
384     volatile struct usb_setup_packet *pkt = (volatile struct usb_setup_packet *) &usb_dpram
385             ->setup_packet;
386     uint8_t req_direction = pkt->bmRequestType;
387     uint8_t req = pkt->bRequest;
388
389     // 将 EP0 IN 的 PID 重置为 1
390     usb_get_endpoint_configuration(EP0_IN_ADDR)->next_pid = 1u;
391
392     if (req_direction == USB_DIR_OUT) {
393         if (req == USB_REQUEST_SET_ADDRESS) {
394             usb_set_device_address(pkt);
395         } else if (req == USB_REQUEST_SET_CONFIGURATION) {
396             usb_set_device_configuration(pkt);
397         } else {
398             usb_acknowledge_out_request();
399             printf("其他 OUT 请求 (0x%x)\r\n", pkt->bRequest);
400         }
401     } else if (req_direction == USB_DIR_IN) {
402         if (req == USB_REQUEST_GET_DESCRIPTOR) {
403             uint16_t descriptor_type = pkt->wValue >> 8;
404
405             switch (descriptor_type) {
406                 case USB_DT_DEVICE:
407                     usb_handle_device_descriptor(pkt);
408                     printf("GET DEVICE DESCRIPTOR\r\n");
409                     break;
410
411                 case USB_DT_CONFIG:
412                     usb_handle_config_descriptor(pkt);
413                     printf("GET CONFIG DESCRIPTOR\r\n");
414                     break;
415
416                 case USB_DT_STRING:
417                     usb_handle_string_descriptor(pkt);
418                     printf("GET STRING DESCRIPTOR\r\n");
419                     break;
420
421                 default:
422                     printf("未处理的 GET_DESCRIPTOR 类型 0x%x\r\n", descriptor_type);
423             }
424         } else {
425             printf("其他 IN 请求 (0x%x)\r\n", pkt->bRequest);
426         }
427     }

```

```

425      }
426    }
427 }
```

4.1.3.2.4. Replying to a setup packet on EP0 IN

The first thing a host will request is the device descriptor, the following code handles that setup request.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 266 - 273

```

266 void usb_handle_device_descriptor(volatile struct usb_setup_packet *pkt) {
267   const struct usb_device_descriptor *d = dev_config.device_descriptor;
268   // EP0 in
269   struct usb_endpoint_configuration *ep = usb_get_endpoint_configuration(EP0_IN_ADDR);
270   // Always respond with pid 1
271   ep->next_pid = 1;
272   usb_start_transfer(ep, (uint8_t *) d, MIN(sizeof(struct usb_device_descriptor), pkt-
>wLength));
273 }
```

The `usb_start_transfer` function copies the data to send into the appropriate hardware buffer, and configures the buffer control register. Once the buffer control register has been written to, the device controller will respond to the host with the data. Before this point, the device will reply with a **NAK**.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c Lines 238 - 260

```

238 void usb_start_transfer(struct usb_endpoint_configuration *ep, uint8_t *buf, uint16_t len) {
239   // We are asserting that the length is <= 64 bytes for simplicity of the example.
240   // For multi packet transfers see the tinyusb port.
241   assert(len <= 64);
242
243   printf("Start transfer of len %d on ep addr 0x%x\n", len, ep->descriptor-
>bEndpointAddress);
244
245   // Prepare buffer control register value
246   uint32_t val = len | USB_BUF_CTRL_AVAIL;
247
248   if (ep_is_tx(ep)) {
249     // Need to copy the data from the user buffer to the usb memory
250     memcpy((void *) ep->data_buffer, (void *) buf, len);
251     // Mark as full
252     val |= USB_BUF_CTRL_FULL;
253   }
254
255   // Set pid and flip for next transfer
256   val |= ep->next_pid ? USB_BUF_CTRL_DATA1_PID : USB_BUF_CTRL_DATA0_PID;
257   ep->next_pid ^= 1u;
258
259   *ep->buffer_control = val;
260 }
```

4.1.4. List of Registers

The USB registers start at a base address of **0x50110000** (defined as `USBCTRL_REGS_BASE` in SDK).

```

425      }
426  }
427 }
```

4.1.3.2.4 在 EP0 IN 端点响应设置包

主机首先请求设备描述符，以下代码处理该设置请求。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第266至273行

```

266 void usb_handle_device_descriptor(volatile struct usb_setup_packet *pkt) {
267     const struct usb_device_descriptor *d = dev_config.device_descriptor;
268     // EP0 输入端点
269     struct usb_endpoint_configuration *ep = usb_get_endpoint_configuration(EP0_IN_ADDR);
270     // 始终响应 PID 1
271     ep->next_pid = 1;
272     usb_start_transfer(ep, (uint8_t *) d, MIN(sizeof(struct usb_device_descriptor), pkt-
>wLength));
273 }
```

`usb_start_transfer` 函数将要发送的数据复制至相应的硬件缓冲区，并配置缓冲区控制寄存器。缓冲区控制寄存器一旦被写入，设备控制器将向主机发送响应数据。在此之前，设备将回复NAK。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/usb/device/dev_lowlevel/dev_lowlevel.c 第238-260行

```

238 void usb_start_transfer(struct usb_endpoint_configuration *ep, uint8_t *buf, uint16_t len) {
239     // 我们断言长度不超过64字节，以简化示例。
240     // 有关多包传输，请参见 tinyusb 端口。
241     assert(len <= 64);
242
243     printf("开始传输，长度 %d，端点地址 0x%x\n", len, ep->descriptor->bEndpointAddress);
244
245     // 准备缓冲区控制寄存器的值
246     uint32_t val = len | USB_BUF_CTRL_AVAIL;
247
248     if (ep_is_tx(ep)) {
249         // 需将数据从用户缓冲区复制至USB内存
250         memcpy((void *) ep->data_buffer, (void *) buf, len);
251         // 标记为已满
252         val |= USB_BUF_CTRL_FULL;
253     }
254
255     // 设置pid并翻转，以便进行下一次传输
256     val |= ep->next_pid ? USB_BUF_CTRL_DATA1_PID : USB_BUF_CTRL_DATA0_PID;
257     ep->next_pid ^= 1u;
258
259     *ep->buffer_control = val;
260 }
```

4.1.4. 寄存器列表

USB寄存器起始基址为 **0x50110000**（在SDK中定义为USBCTRL_REGS_BASE）。

Table 397. List of USB registers

Offset	Name	Info
0x00	ADDR_ENDP	Device address and endpoint control
0x04	ADDR_ENDP1	Interrupt endpoint 1. Only valid for HOST mode.
0x08	ADDR_ENDP2	Interrupt endpoint 2. Only valid for HOST mode.
0x0c	ADDR_ENDP3	Interrupt endpoint 3. Only valid for HOST mode.
0x10	ADDR_ENDP4	Interrupt endpoint 4. Only valid for HOST mode.
0x14	ADDR_ENDP5	Interrupt endpoint 5. Only valid for HOST mode.
0x18	ADDR_ENDP6	Interrupt endpoint 6. Only valid for HOST mode.
0x1c	ADDR_ENDP7	Interrupt endpoint 7. Only valid for HOST mode.
0x20	ADDR_ENDP8	Interrupt endpoint 8. Only valid for HOST mode.
0x24	ADDR_ENDP9	Interrupt endpoint 9. Only valid for HOST mode.
0x28	ADDR_ENDP10	Interrupt endpoint 10. Only valid for HOST mode.
0x2c	ADDR_ENDP11	Interrupt endpoint 11. Only valid for HOST mode.
0x30	ADDR_ENDP12	Interrupt endpoint 12. Only valid for HOST mode.
0x34	ADDR_ENDP13	Interrupt endpoint 13. Only valid for HOST mode.
0x38	ADDR_ENDP14	Interrupt endpoint 14. Only valid for HOST mode.
0x3c	ADDR_ENDP15	Interrupt endpoint 15. Only valid for HOST mode.
0x40	MAIN_CTRL	Main control register
0x44	SOF_WR	Set the SOF (Start of Frame) frame number in the host controller. The SOF packet is sent every 1ms and the host will increment the frame number by 1 each time.
0x48	SOF_RD	Read the last SOF (Start of Frame) frame number seen. In device mode the last SOF received from the host. In host mode the last SOF sent by the host.
0x4c	SIE_CTRL	SIE control register
0x50	SIE_STATUS	SIE status register
0x54	INT_EP_CTRL	interrupt endpoint control register
0x58	BUFF_STATUS	Buffer status register. A bit set here indicates that a buffer has completed on the endpoint (if the buffer interrupt is enabled). It is possible for 2 buffers to be completed, so clearing the buffer status bit may instantly re set it on the next clock cycle.
0x5c	BUFF_CPU_SHOULD_HANDLE	Which of the double buffers should be handled. Only valid if using an interrupt per buffer (i.e. not per 2 buffers). Not valid for host interrupt endpoint polling because they are only single buffered.
0x60	EP_ABORT	Device only: Can be set to ignore the buffer control register for this endpoint in case you would like to revoke a buffer. A NAK will be sent for every access to the endpoint until this bit is cleared. A corresponding bit in EP_ABORT_DONE is set when it is safe to modify the buffer control register.

表397. USB
寄存器列表

偏移量	名称	说明
0x00	ADDR_ENDP	设备地址及端点控制
0x04	ADDR_ENDP1	中断端点1。仅适用于HOST模式。
0x08	ADDR_ENDP2	中断端点2。仅适用于HOST模式。
0x0c	ADDR_ENDP3	中断端点3。仅适用于HOST模式。
0x10	ADDR_ENDP4	中断端点4。仅适用于HOST模式。
0x14	ADDR_ENDP5	中断端点5。仅适用于HOST模式。
0x18	ADDR_ENDP6	中断端点6。仅适用于HOST模式。
0x1c	ADDR_ENDP7	中断端点7。仅适用于HOST模式。
0x20	ADDR_ENDP8	中断端点8。仅适用于HOST模式。
0x24	ADDR_ENDP9	中断端点9。仅适用于HOST模式。
0x28	ADDR_ENDP10	中断端点10。仅适用于HOST模式。
0x2c	ADDR_ENDP11	中断端点11。仅适用于HOST模式。
0x30	ADDR_ENDP12	中断端点12。仅适用于HOST模式。
0x34	ADDR_ENDP13	中断端点13。仅适用于HOST模式。
0x38	ADDR_ENDP14	中断端点14。仅适用于HOST模式。
0x3c	ADDR_ENDP15	中断端点15。仅适用于HOST模式。
0x40	MAIN_CTRL	主控制寄存器
0x44	SOF_WR	设置主控制器中的 SOF（帧开始）帧编号。 SOF 数据包每 1 毫秒发送一次，主机每次将帧编号递增 1。
0x48	SOF_RD	读取最近接收到的 SOF（帧开始）帧编号。设备模式下，最后接收到的来自主机的 SOF 帧。主机模式下，最后由主机发送的 SOF 帧。
0x4c	SIE_CTRL	SIE 控制寄存器
0x50	SIE_STATUS	SIE 状态寄存器
0x54	INT_EP_CTRL	中断端点控制寄存器
0x58	BUFF_STATUS	缓冲区状态寄存器。此位置位表示端点上的缓冲区已完成（前提是缓冲区中断已启用）。有可能两个缓冲区同时完成，因此清除缓冲区状态位后，下一时钟周期可能会立即重新置位。
0x5c	BUFF_CPU_SHOULD_HANDLE	应处理的双缓冲区中的哪一个。仅在每缓冲区产生中断时有效（即非每两个缓冲区）。对主机中断端点轮询无效，因为它们仅为单缓冲区。
0x60	EP_ABORT	仅限设备：可设置为忽略该端点的缓冲区控制寄存器，以便撤销缓冲区。在清除此位之前，每次访问该端点时均会发送NAK。当可安全修改缓冲区控制寄存器时，EP_ABORT_DONE中对应位将被置位。

Offset	Name	Info
0x64	EP_ABORT_DONE	Device only: Used in conjunction with EP_ABORT. Set once an endpoint is idle so the programmer knows it is safe to modify the buffer control register.
0x68	EP_STALL_ARM	Device: this bit must be set in conjunction with the STALL bit in the buffer control register to send a STALL on EP0. The device controller clears these bits when a SETUP packet is received because the USB spec requires that a STALL condition is cleared when a SETUP packet is received.
0x6c	NAK_POLL	Used by the host controller. Sets the wait time in microseconds before trying again if the device replies with a NAK.
0x70	EP_STATUS_STALL_NAK	Device: bits are set when the IRQ_ON_NAK or IRQ_ON_STALL bits are set. For EP0 this comes from SIE_CTRL. For all other endpoints it comes from the endpoint control register.
0x74	USB_MUXING	Where to connect the USB controller. Should be to_phy by default.
0x78	USB_PWR	Overrides for the power signals in the event that the VBUS signals are not hooked up to GPIO. Set the value of the override and then the override enable to switch over to the override value.
0x7c	USBPHY_DIRECT	This register allows for direct control of the USB phy. Use in conjunction with usbphy_direct_override register to enable each override bit.
0x80	USBPHY_DIRECT_OVERRIDE	Override enable for each control in usbphy_direct
0x84	USBPHY_TRIM	Used to adjust trim values of USB phy pull down resistors.
0x8c	INTR	Raw Interrupts
0x90	INTE	Interrupt Enable
0x94	INTF	Interrupt Force
0x98	INTS	Interrupt status after masking & forcing

USB: ADDR_ENDP Register

Offset: 0x00

Description

Device address and endpoint control

Table 398.
ADDR_ENDP Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19:16	ENDPOINT: Device endpoint to send data to. Only valid for HOST mode.	RW	0x0
15:7	Reserved.	-	-
6:0	ADDRESS: In device mode, the address that the device should respond to. Set in response to a SET_ADDR setup packet from the host. In host mode set to the address of the device to communicate with.	RW	0x00

USB: ADDR_ENDP1, ADDR_ENDP2, ..., ADDR_ENDP14, ADDR_ENDP15 Registers

偏移量	名称	说明
0x64	EP_ABORT_DONE	仅限设备：与 EP_ABORT 配合使用。端点空闲时设置此位，以通知程序员可安全修改缓冲区控制寄存器。
0x68	EP_STALL_ARM	设备：此位须与缓冲区控制寄存器中的 STALL 位同时设置，以便在 EP0 端点发送 STALL。设备控制器在收到 SETUP 包时会清除此位，因为 USB 规范要求在收到 SETUP 包时清除 STALL 状态。
0x6c	NAK_POLL	由主机控制器使用。设置设备响应 NAK 时重试前的等待时间，单位为微秒。
0x70	EP_STATUS_STALL_NAK	设备：当 IRQ_ON_NAK 或 IRQ_ON_STALL 位被置位时，此位亦被置位。对于 EP0，该信号由 SIE_CTRL 提供。对于所有其他端点，其来源于端点控制寄存器。
0x74	USB_MUXING	USB 控制器的连接位置。默认应为 to_phy。
0x78	USB_PWR	当 VBUS 信号未连接至 GPIO 时，用于电源信号的覆盖。先设置覆盖值，再启用覆盖以切换至该值。
0x7c	USBPHY_DIRECT	此寄存器允许直接控制 USB 物理层。需配合 usbphy_direct_override 寄存器使用以启用各覆盖位。
0x80	USBPHY_DIRECT_OVERRIDE	usbphy_direct 中各控制项的覆盖使能。
0x84	USBPHY_TRIM	用于调整 USB 物理层下拉电阻的调节值。
0x8c	INTR	原始中断
0x90	INTE	中断使能
0x94	INTF	中断强制
0x98	INTS	掩码及强制后的中断状态

USB: ADDR_ENDP 寄存器

偏移: 0x00

描述

设备地址及端点控制

表 398。
ADDR_ENDP 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:16	ENDPOINT : 设备端点，用于发送数据。仅在主机模式下有效。	读写	0x0
15:7	保留。	-	-
6:0	ADDRESS : 设备模式下设备应响应的地址，根据主机发送的 SET_ADDR 请求包设置。主机模式下设置为待通信设备的地址。	读写	0x00

USB: ADDR_ENDP1、ADDR_ENDP2、...、ADDR_ENDP14、ADDR_ENDP15 寄存器

Offsets: 0x04, 0x08, ..., 0x38, 0x3c

Description

Interrupt endpoint N. Only valid for HOST mode.

Table 399.
ADDR_ENDP1,
ADDR_ENDP2, ...,
ADDR_ENDP14,
Registers
ADDR_ENDP15

Bits	Description	Type	Reset
31:27	Reserved.	-	-
26	INTEP_PREAMBLE: Interrupt EP requires preamble (is a low speed device on a full speed hub)	RW	0x0
25	INTEP_DIR: Direction of the interrupt endpoint. In=0, Out=1	RW	0x0
24:20	Reserved.	-	-
19:16	ENDPOINT: Endpoint number of the interrupt endpoint	RW	0x0
15:7	Reserved.	-	-
6:0	ADDRESS: Device address	RW	0x00

USB: MAIN_CTRL Register

Offset: 0x40

Description

Main control register

Table 400.
MAIN_CTRL Register

Bits	Description	Type	Reset
31	SIM_TIMING: Reduced timings for simulation	RW	0x0
30:2	Reserved.	-	-
1	HOST_NDEVICE: Device mode = 0, Host mode = 1	RW	0x0
0	CONTROLLER_EN: Enable controller	RW	0x0

USB: SOF_WR Register

Offset: 0x44

Description

Set the SOF (Start of Frame) frame number in the host controller. The SOF packet is sent every 1ms and the host will increment the frame number by 1 each time.

Table 401. SOF_WR Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-
10:0	COUNT	WF	0x000

USB: SOF_RD Register

Offset: 0x48

Description

Read the last SOF (Start of Frame) frame number seen. In device mode the last SOF received from the host. In host mode the last SOF sent by the host.

Table 402. SOF_RD Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-

偏移: 0x04, 0x08, ..., 0x38, 0x3c

描述

中断端点 N。仅适用于HOST模式。

表399。
ADDR_ENDP1,
ADDR_ENDP2, ...,
ADDR_ENDP14,
ADDR_ENDP15
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26	INTEP_PREAMBLE : 中断端点需带前导码 (在全速集线器上的低速设备)	读写	0x0
25	INTEP_DIR : 中断端点方向。输入=0, 输出=1	读写	0x0
24:20	保留。	-	-
19:16	ENDPOINT : 中断端点编号	读写	0x0
15:7	保留。	-	-
6:0	ADDRESS : 设备地址	读写	0x00

USB: MAIN_CTRL 寄存器

偏移: 0x40

描述

主控制寄存器

表400。
MAIN_CTRL 寄存器

位	描述	类型	复位值
31	SIM_TIMING : 仿真用缩减时序	读写	0x0
30:2	保留。	-	-
1	HOST_NDEVICE : 设备模式=0, 主机模式=1	读写	0x0
0	CONTROLLER_EN : 启用控制器	读写	0x0

USB: SOF_WR 寄存器

偏移: 0x44

描述

设置主机控制器中的SOF (帧起始) 帧编号。SOF 数据包每 1 毫秒发送一次，主机每次将帧编号递增 1。

表 401. SOF_WR
寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10:0	计数	WF	0x000

USB: SOF_RD 寄存器

偏移: 0x48

描述

读取最近接收到的 SOF (帧开始) 帧编号。设备模式下，最后接收到的来自主机的 SOF 帧。主机模式下，最后由主机发送的 SOF 帧。

表 402. SOF_RD
寄存器

位	描述	类型	复位值
31:11	保留。	-	-

Bits	Description	Type	Reset
10:0	COUNT	RO	0x000

USB: SIE_CTRL Register

Offset: 0x4c

Description

SIE control register

Table 403. SIE_CTRL Register

Bits	Description	Type	Reset
31	EP0_INT_STALL : Device: Set bit in EP_STATUS_STALL_NAK when EP0 sends a STALL	RW	0x0
30	EP0_DOUBLE_BUF : Device: EP0 single buffered = 0, double buffered = 1	RW	0x0
29	EP0_INT_1BUF : Device: Set bit in BUFF_STATUS for every buffer completed on EP0	RW	0x0
28	EP0_INT_2BUF : Device: Set bit in BUFF_STATUS for every 2 buffers completed on EP0	RW	0x0
27	EP0_INT_NAK : Device: Set bit in EP_STATUS_STALL_NAK when EP0 sends a NAK	RW	0x0
26	DIRECT_EN : Direct bus drive enable	RW	0x0
25	DIRECT_DP : Direct control of DP	RW	0x0
24	DIRECT_DM : Direct control of DM	RW	0x0
23:19	Reserved.	-	-
18	TRANSCEIVER_PD : Power down bus transceiver	RW	0x0
17	RPU_OPT : Device: Pull-up strength (0=1k2, 1=2k3)	RW	0x0
16	PULLUP_EN : Device: Enable pull up resistor	RW	0x0
15	PULLDOWN_EN : Host: Enable pull down resistors	RW	0x0
14	Reserved.	-	-
13	RESET_BUS : Host: Reset bus	SC	0x0
12	RESUME : Device: Remote wakeup. Device can initiate its own resume after suspend.	SC	0x0
11	VBUS_EN : Host: Enable VBUS	RW	0x0
10	KEEP_ALIVE_EN : Host: Enable keep alive packet (for low speed bus)	RW	0x0
9	SOF_EN : Host: Enable SOF generation (for full speed bus)	RW	0x0
8	SOF_SYNC : Host: Delay packet(s) until after SOF	RW	0x0
7	Reserved.	-	-
6	PREAMBLE_EN : Host: Preamble enable for LS device on FS hub	RW	0x0
5	Reserved.	-	-
4	STOP_TRANS : Host: Stop transaction	SC	0x0
3	RECEIVE_DATA : Host: Receive transaction (IN to host)	RW	0x0
2	SEND_DATA : Host: Send transaction (OUT from host)	RW	0x0

位	描述	类型	复位值
10:0	计数	只读	0x000

USB：SIE_CTRL 寄存器

偏移：0x4c

描述

SIE 控制寄存器

表 403. SIE_CTRL 寄存器

位	描述	类型	复位值
31	EPO_INT_STALL : 设备：当 EP0 发送 STALL 时，设置 EP_STATUS_STALL_NAK 中相应位	读写	0x0
30	EPO_DOUBLE_BUF : 设备：EP0 单缓冲区 = 0，双缓冲区 = 1	读写	0x0
29	EPO_INT_1BUF : 设备：每完成 EP0 单个缓冲区时，设置 BUFF_STATUS 中相 应位	读写	0x0
28	EPO_INT_2BUF : 设备：每完成 EP0 两个缓冲区时，设置 BUFF_STATUS 中相 应位	读写	0x0
27	EPO_INT_NAK : 设备：当 EP0 发送 NAK 时，设置 EP_STATUS_STALL_NAK 中相 应位	读写	0x0
26	DIRECT_EN : 直接总线驱动使能	读写	0x0
25	DIRECT_DP : 直接控制 DP	读写	0x0
24	DIRECT_DM : 直接控制 DM	读写	0x0
23:19	保留。	-	-
18	TRANSCEIVER_PD : 断电总线收发器	读写	0x0
17	RPU_OPT : 设备：上拉电阻强度 (0=1K2, 1=2k3)	读写	0x0
16	PULLUP_EN : 设备：启用上拉电阻	读写	0x0
15	PULLDOWN_EN : 主机：启用下拉电阻	读写	0x0
14	保留。	-	-
13	RESET_BUS : 主机：复位总线	SC	0x0
12	RESUME : 设备：远程唤醒设备可在挂起后自发恢复。	SC	0x0
11	VBUS_EN : 主机：启用 VBUS	读写	0x0
10	KEEP_ALIVE_EN : 主机：启用保持活动包（针对低速总线）	读写	0x0
9	SOF_EN : 主机：启用 SOF 生成（针对全速总线）	读写	0x0
8	SOF_SYNC : 主机：延迟数据包至 SOF 后	读写	0x0
7	保留。	-	-
6	PREAMBLE_EN : 主机：为全速集线器上的低速设备启用前导码	读写	0x0
5	保留。	-	-
4	STOP_TRANS : 主机：停止事务	SC	0x0
3	RECEIVE_DATA : 主机：接收事务 (IN 至主机)	读写	0x0
2	SEND_DATA : 主机：发送事务 (OUT 从主机)	读写	0x0

Bits	Description	Type	Reset
1	SEND_SETUP : Host: Send Setup packet	RW	0x0
0	START_TRANS : Host: Start transaction	SC	0x0

USB: SIE_STATUS Register

Offset: 0x50

Description

SIE status register

Table 404.
SIE_STATUS Register

Bits	Description	Type	Reset
31	DATA_SEQ_ERROR : Data Sequence Error. The device can raise a sequence error in the following conditions: * A SETUP packet is received followed by a DATA1 packet (data phase should always be DATA0) * An OUT packet is received from the host but doesn't match the data pid in the buffer control register read from DPRAM The host can raise a data sequence error in the following conditions: * An IN packet from the device has the wrong data PID	WC	0x0
30	ACK_REC : ACK received. Raised by both host and device.	WC	0x0
29	STALL_REC : Host: STALL received	WC	0x0
28	NAK_REC : Host: NAK received	WC	0x0
27	RX_TIMEOUT : RX timeout is raised by both the host and device if an ACK is not received in the maximum time specified by the USB spec.	WC	0x0
26	RX_OVERFLOW : RX overflow is raised by the Serial RX engine if the incoming data is too fast.	WC	0x0
25	BIT_STUFF_ERROR : Bit Stuff Error. Raised by the Serial RX engine.	WC	0x0
24	CRC_ERROR : CRC Error. Raised by the Serial RX engine.	WC	0x0
23:20	Reserved.	-	-
19	BUS_RESET : Device: bus reset received	WC	0x0
18	TRANS_COMPLETE : Transaction complete. Raised by device if: * An IN or OUT packet is sent with the LAST_BUFF bit set in the buffer control register Raised by host if: * A setup packet is sent when no data in or data out transaction follows * An IN packet is received and the LAST_BUFF bit is set in the buffer control register * An IN packet is received with zero length * An OUT packet is sent and the LAST_BUFF bit is set	WC	0x0
17	SETUP_REC : Device: Setup packet received	WC	0x0

位	描述	类型	复位值
1	SEND_SETUP : 主机: 发送设置包	读写	0x0
0	START_TRANS : 主机: 启动事务	SC	0x0

USB: SIE_STATUS 寄存器

偏移: 0x50

描述

SIE 状态寄存器

表 404。
SIE_STATUS 寄存器

位	描述	类型	复位值
31	DATA_SEQ_ERROR : 数据序列错误。 设备在以下情况下可能触发序列错误： * 接收到一个 SETUP 包后紧跟 DATA1 包 (数据阶段应始终为 DATA0) * 从主机接收到的 OUT 包其数据 PID 与从 DPRAM 读取的缓冲区控制寄存器中的数据 PID 不匹配 主机在以下情况下可能触发数据序列错误： * 设备发送的 IN 包数据 PID 错误	WC	0x0
30	ACK_REC : 已接收 ACK。由主机和设备共同触发。	WC	0x0
29	STALL_REC : 主机: 已接收 STALL	WC	0x0
28	NAK_REC : 主机: 已接收 NAK	WC	0x0
27	RX_TIMEOUT : 若在 USB 规范规定的最长时间内未接收到 ACK, 主机和设备均会触发接收超时。	WC	0x0
26	RX_OVERFLOW : 当接收数据速率过快时, 串行接收引擎触发接收溢出。	WC	0x0
25	BIT_STUFF_ERROR : 位填充错误。由串行接收引擎触发。	WC	0x0
24	CRC_ERROR : CRC 校验错误。由串行接收引擎触发。	WC	0x0
23:20	保留。	-	-
19	BUS_RESET : 设备: 接收到总线复位信号	WC	0x0
18	TRANS_COMPLETE : 事务完成。 由设备触发, 条件如下： * 当 IN 包或 OUT 包发送时, 缓冲区控制寄存器中的 LAST_BUFF 位被设置	WC	0x0
	由主机触发, 条件如下： * 发送设置包后未跟随数据输入或数据输出事务 * 接收到 IN 包且缓冲区控制寄存器中的 LAST_BUFF 位被设置 * 收到长度为零的 IN 包 * 发送 OUT 包且缓冲区控制寄存器中的 LAST_BUFF 位被设置		
17	SETUP_REC : 设备: 收到设置包	WC	0x0

Bits	Description	Type	Reset
16	CONNECTED : Device: connected	WC	0x0
15:12	Reserved.	-	-
11	RESUME : Host: Device has initiated a remote resume. Device: host has initiated a resume.	WC	0x0
10	VBUS_OVER_CURR : VBUS over current detected	RO	0x0
9:8	SPEED : Host: device speed. Disconnected = 00, LS = 01, FS = 10	WC	0x0
7:5	Reserved.	-	-
4	SUSPENDED : Bus in suspended state. Valid for device and host. Host and device will go into suspend if neither Keep Alive / SOF frames are enabled.	WC	0x0
3:2	LINE_STATE : USB bus line state	RO	0x0
1	Reserved.	-	-
0	VBUS_DETECTED : Device: VBUS Detected	RO	0x0

USB: INT_EP_CTRL Register

Offset: 0x54

Description

interrupt endpoint control register

Table 405.
INT_EP_CTRL Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:1	INT_EP_ACTIVE : Host: Enable interrupt endpoint 1 → 15	RW	0x0000
0	Reserved.	-	-

USB: BUFF_STATUS Register

Offset: 0x58

Description

Buffer status register. A bit set here indicates that a buffer has completed on the endpoint (if the buffer interrupt is enabled). It is possible for 2 buffers to be completed, so clearing the buffer status bit may instantly re set it on the next clock cycle.

Table 406.
BUFF_STATUS Register

Bits	Description	Type	Reset
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0

位	描述	类型	复位值
16	CONNECTED : 设备: 已连接	WC	0x0
15:12	保留。	-	-
11	RESUME : 主机: 设备已发起远程恢复。设备: 主机已发起恢复。	WC	0x0
10	VBUS_OVER_CURR : 检测到VBUS过流	只读	0x0
9:8	SPEED : 主机: 设备速度。断开连接 = 00, 低速 = 01, 全速 = 10	WC	0x0
7:5	保留。	-	-
4	SUSPENDED : 总线处于挂起状态。适用于设备和主机。若未启用Keep Alive或SOF帧，则主机和设备将进入挂起状态。	WC	0x0
3:2	LINE_STATE : USB总线线路状态	只读	0x0
1	保留。	-	-
0	VBUS_DETECTED : 设备: 检测到VBUS	只读	0x0

USB: INT_EP_CTRL寄存器

偏移: 0x54

说明

中断端点控制寄存器

表405。
INT_EP_CTRL寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:1	INT_EP_ACTIVE : 主机: 启用中断端点1 → 15	读写	0x0000
0	保留。	-	-

USB: BUFF_STATUS寄存器

偏移: 0x58

说明

缓冲区状态寄存器。此位置位表示端点上的缓冲区已完成（前提是缓冲区中断已启用）。有可能两个缓冲区同时完成，因此清除缓冲区状态位后，下一时钟周期可能会立即重新置位。

表406。
BUFF_STATUS
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0

Bits	Description	Type	Reset
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: BUFF_CPU_SHOULD_HANDLE Register

Offset: 0x5c

Description

Which of the double buffers should be handled. Only valid if using an interrupt per buffer (i.e. not per 2 buffers). Not valid for host interrupt endpoint polling because they are only single buffered.

Table 407.
BUFF_CPU_SHOULD_HANDLE Register

Bits	Description	Type	Reset
31	EP15_OUT	RO	0x0
30	EP15_IN	RO	0x0
29	EP14_OUT	RO	0x0
28	EP14_IN	RO	0x0
27	EP13_OUT	RO	0x0
26	EP13_IN	RO	0x0

位	描述	类型	复位值
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB：BUFF_CPU_SHOULD_HANDLE寄存器

偏移：0x5c

描述

应处理的双缓冲区中的哪一个。仅在每缓冲区产生中断时有效（即非每两个缓冲区）。对主机中断端点轮询无效，因为它们仅为单缓冲区。

表407。
BUFF_CPU_SHOULD_HANDLE寄存器

位	描述	类型	复位值
31	EP15_OUT	只读	0x0
30	EP15_IN	只读	0x0
29	EP14_OUT	只读	0x0
28	EP14_IN	只读	0x0
27	EP13_OUT	只读	0x0
26	EP13_IN	只读	0x0

Bits	Description	Type	Reset
25	EP12_OUT	RO	0x0
24	EP12_IN	RO	0x0
23	EP11_OUT	RO	0x0
22	EP11_IN	RO	0x0
21	EP10_OUT	RO	0x0
20	EP10_IN	RO	0x0
19	EP9_OUT	RO	0x0
18	EP9_IN	RO	0x0
17	EP8_OUT	RO	0x0
16	EP8_IN	RO	0x0
15	EP7_OUT	RO	0x0
14	EP7_IN	RO	0x0
13	EP6_OUT	RO	0x0
12	EP6_IN	RO	0x0
11	EP5_OUT	RO	0x0
10	EP5_IN	RO	0x0
9	EP4_OUT	RO	0x0
8	EP4_IN	RO	0x0
7	EP3_OUT	RO	0x0
6	EP3_IN	RO	0x0
5	EP2_OUT	RO	0x0
4	EP2_IN	RO	0x0
3	EP1_OUT	RO	0x0
2	EP1_IN	RO	0x0
1	EP0_OUT	RO	0x0
0	EP0_IN	RO	0x0

USB: EP_ABORT Register

Offset: 0x60

Description

Device only: Can be set to ignore the buffer control register for this endpoint in case you would like to revoke a buffer. A NAK will be sent for every access to the endpoint until this bit is cleared. A corresponding bit in [EP_ABORT_DONE](#) is set when it is safe to modify the buffer control register.

Table 408. EP_ABORT Register

Bits	Description	Type	Reset
31	EP15_OUT	RW	0x0
30	EP15_IN	RW	0x0
29	EP14_OUT	RW	0x0

位	描述	类型	复位值
25	EP12_OUT	只读	0x0
24	EP12_IN	只读	0x0
23	EP11_OUT	只读	0x0
22	EP11_IN	只读	0x0
21	EP10_OUT	只读	0x0
20	EP10_IN	只读	0x0
19	EP9_OUT	只读	0x0
18	EP9_IN	只读	0x0
17	EP8_OUT	只读	0x0
16	EP8_IN	只读	0x0
15	EP7_OUT	只读	0x0
14	EP7_IN	只读	0x0
13	EP6_OUT	只读	0x0
12	EP6_IN	只读	0x0
11	EP5_OUT	只读	0x0
10	EP5_IN	只读	0x0
9	EP4_OUT	只读	0x0
8	EP4_IN	只读	0x0
7	EP3_OUT	只读	0x0
6	EP3_IN	只读	0x0
5	EP2_OUT	只读	0x0
4	EP2_IN	只读	0x0
3	EP1_OUT	只读	0x0
2	EP1_IN	只读	0x0
1	EP0_OUT	只读	0x0
0	EP0_IN	只读	0x0

USB: EP_ABORT 寄存器

偏移: 0x60

描述

仅限设备：可设置为忽略该端点的缓冲区控制寄存器，以便撤销缓冲区。在清除此位之前，每次访问该端点时均会发送NAK。当可以安全修改缓冲区控制寄存器时，EP_ABORT_DONE的对应位将被置位。

表 408. EP_ABORT 寄存器

位	描述	类型	复位值
31	EP15_OUT	读写	0x0
30	EP15_IN	读写	0x0
29	EP14_OUT	读写	0x0

Bits	Description	Type	Reset
28	EP14_IN	RW	0x0
27	EP13_OUT	RW	0x0
26	EP13_IN	RW	0x0
25	EP12_OUT	RW	0x0
24	EP12_IN	RW	0x0
23	EP11_OUT	RW	0x0
22	EP11_IN	RW	0x0
21	EP10_OUT	RW	0x0
20	EP10_IN	RW	0x0
19	EP9_OUT	RW	0x0
18	EP9_IN	RW	0x0
17	EP8_OUT	RW	0x0
16	EP8_IN	RW	0x0
15	EP7_OUT	RW	0x0
14	EP7_IN	RW	0x0
13	EP6_OUT	RW	0x0
12	EP6_IN	RW	0x0
11	EP5_OUT	RW	0x0
10	EP5_IN	RW	0x0
9	EP4_OUT	RW	0x0
8	EP4_IN	RW	0x0
7	EP3_OUT	RW	0x0
6	EP3_IN	RW	0x0
5	EP2_OUT	RW	0x0
4	EP2_IN	RW	0x0
3	EP1_OUT	RW	0x0
2	EP1_IN	RW	0x0
1	EP0_OUT	RW	0x0
0	EP0_IN	RW	0x0

USB: EP_ABORT_DONE Register

Offset: 0x64

Description

Device only. Used in conjunction with EP_ABORT. Set once an endpoint is idle so the programmer knows it is safe to modify the buffer control register.

位	描述	类型	复位值
28	EP14_IN	读写	0x0
27	EP13_OUT	读写	0x0
26	EP13_IN	读写	0x0
25	EP12_OUT	读写	0x0
24	EP12_IN	读写	0x0
23	EP11_OUT	读写	0x0
22	EP11_IN	读写	0x0
21	EP10_OUT	读写	0x0
20	EP10_IN	读写	0x0
19	EP9_OUT	读写	0x0
18	EP9_IN	读写	0x0
17	EP8_OUT	读写	0x0
16	EP8_IN	读写	0x0
15	EP7_OUT	读写	0x0
14	EP7_IN	读写	0x0
13	EP6_OUT	读写	0x0
12	EP6_IN	读写	0x0
11	EP5_OUT	读写	0x0
10	EP5_IN	读写	0x0
9	EP4_OUT	读写	0x0
8	EP4_IN	读写	0x0
7	EP3_OUT	读写	0x0
6	EP3_IN	读写	0x0
5	EP2_OUT	读写	0x0
4	EP2_IN	读写	0x0
3	EP1_OUT	读写	0x0
2	EP1_IN	读写	0x0
1	EP0_OUT	读写	0x0
0	EP0_IN	读写	0x0

USB：EP_ABORT_DONE 寄存器

偏移：0x64

描述

仅限设备：与 EP_ABORT 配合使用。端点空闲时设置此位，以通知程序员可安全修改缓冲区控制寄存器。

Table 409.
EP_ABORT_DONE
Register

Bits	Description	Type	Reset
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: EP_STALL_ARM Register

Offset: 0x68

表 409。
EP_ABORT_DONE
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: EP_STALL_ARM 寄存器

偏移量: 0x68

Description

Device: this bit must be set in conjunction with the **STALL** bit in the buffer control register to send a STALL on EP0. The device controller clears these bits when a SETUP packet is received because the USB spec requires that a STALL condition is cleared when a SETUP packet is received.

*Table 410.
EP_STALL_ARM
Register*

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	EP0_OUT	RW	0x0
0	EP0_IN	RW	0x0

USB: NAK_POLL Register**Offset:** 0x6c**Description**

Used by the host controller. Sets the wait time in microseconds before trying again if the device replies with a NAK.

*Table 411. NAK_POLL
Register*

Bits	Description	Type	Reset
31:26	Reserved.	-	-
25:16	DELAY_FS : NAK polling interval for a full speed device	RW	0x010
15:10	Reserved.	-	-
9:0	DELAY_LS : NAK polling interval for a low speed device	RW	0x010

USB: EP_STATUS_STALL_NAK Register**Offset:** 0x70**Description**

Device: bits are set when the **IRQ_ON_NAK** or **IRQ_ON_STALL** bits are set. For EP0 this comes from **SIE_CTRL**. For all other endpoints it comes from the endpoint control register.

*Table 412.
EP_STATUS_STALL_NA
K Register*

Bits	Description	Type	Reset
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0

描述

设备：此位必须与缓冲区控制寄存器中的 **STALL** 位同时设置，以便在 EP0 上发送 STALL 信号。

当接收到 SETUP 数据包时，设备控制器将清除这些位，因为 USB 规范要求在接收到 SETUP 数据包时必须清除 STALL 状态。

表 410。
EP_STALL_ARM
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	EP0_OUT	读写	0x0
0	EP0_IN	读写	0x0

USB：NAK_POLL 寄存器

偏移: 0x6c

描述

由主机控制器使用。设置设备响应 NAK 时重试前的等待时间，单位为微秒。

表 411. NAK_POLL
寄存器

位	描述	类型	复位值
31:26	保留。	-	-
25:16	DELAY_FS ：全速设备的 NAK 轮询间隔	读写	0x010
15:10	保留。	-	-
9:0	DELAY_LS ：低速设备的 NAK 轮询间隔	读写	0x010

USB：EP_STATUS_STALL_NAK 寄存器

偏移: 0x70

描述

设备：当 **IRQ_ON_NAK** 或 **IRQ_ON_STALL** 位被置位时，此位亦被置位。对于 EP0，该信号由 **SIE_CTRL** 提供。对于所有其他端点，其来源于端点控制寄存器。

表 412。
EP_STATUS_STALL_NAK
寄存器

位	描述	类型	复位值
31	EP15_OUT	WC	0x0
30	EP15_IN	WC	0x0
29	EP14_OUT	WC	0x0
28	EP14_IN	WC	0x0
27	EP13_OUT	WC	0x0
26	EP13_IN	WC	0x0
25	EP12_OUT	WC	0x0
24	EP12_IN	WC	0x0
23	EP11_OUT	WC	0x0
22	EP11_IN	WC	0x0
21	EP10_OUT	WC	0x0
20	EP10_IN	WC	0x0
19	EP9_OUT	WC	0x0
18	EP9_IN	WC	0x0

Bits	Description	Type	Reset
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: USB_MUXING Register

Offset: 0x74

Description

Where to connect the USB controller. Should be to_phy by default.

Table 413.
USB_MUXING Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	SOFTCON	RW	0x0
2	TO_DIGITAL_PAD	RW	0x0
1	TO_EXTPHY	RW	0x0
0	TO_PHY	RW	0x0

USB: USB_PWR Register

Offset: 0x78

Description

Overrides for the power signals in the event that the VBUS signals are not hooked up to GPIO. Set the value of the override and then the override enable to switch over to the override value.

Table 414. USB_PWR
Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-

位	描述	类型	复位值
17	EP8_OUT	WC	0x0
16	EP8_IN	WC	0x0
15	EP7_OUT	WC	0x0
14	EP7_IN	WC	0x0
13	EP6_OUT	WC	0x0
12	EP6_IN	WC	0x0
11	EP5_OUT	WC	0x0
10	EP5_IN	WC	0x0
9	EP4_OUT	WC	0x0
8	EP4_IN	WC	0x0
7	EP3_OUT	WC	0x0
6	EP3_IN	WC	0x0
5	EP2_OUT	WC	0x0
4	EP2_IN	WC	0x0
3	EP1_OUT	WC	0x0
2	EP1_IN	WC	0x0
1	EP0_OUT	WC	0x0
0	EP0_IN	WC	0x0

USB: USB_MUXING 寄存器

偏移: 0x74

描述

USB 控制器的连接位置。默认应为 to_phy。

表 413。
USB_MUXING 寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	SOFTCON	读写	0x0
2	TO_DIGITAL_PAD	读写	0x0
1	TO_EXTPHY	读写	0x0
0	TO_PHY	读写	0x0

USB: USB_PWR 寄存器

偏移: 0x78

描述

当 VBUS 信号未连接至 GPIO 时，用于电源信号的覆盖。先设置覆盖值，再启用覆盖以切换至该值。

表 414。USB_PWR
寄存器

位	描述	类型	复位值
31:6	保留。	-	-

Bits	Description	Type	Reset
5	OVERCURR_DETECT_EN	RW	0x0
4	OVERCURR_DETECT	RW	0x0
3	VBUS_DETECT_OVERRIDE_EN	RW	0x0
2	VBUS_DETECT	RW	0x0
1	VBUS_EN_OVERRIDE_EN	RW	0x0
0	VBUS_EN	RW	0x0

USB: USBPHY_DIRECT Register

Offset: 0x7c

Description

This register allows for direct control of the USB phy. Use in conjunction with usbphy_direct_override register to enable each override bit.

Table 415.
USBPHY_DIRECT
Register

Bits	Description	Type	Reset
31:23	Reserved.	-	-
22	DM_OVV : DM over voltage	RO	0x0
21	DP_OVV : DP over voltage	RO	0x0
20	DM_OVCN : DM overcurrent	RO	0x0
19	DP_OVCN : DP overcurrent	RO	0x0
18	RX_DM : DPM pin state	RO	0x0
17	RX_DP : DPP pin state	RO	0x0
16	RX_DD : Differential RX	RO	0x0
15	TX_DIFFMODE : TX_DIFFMODE=0: Single ended mode TX_DIFFMODE=1: Differential drive mode (TX_DM, TX_DM_OE ignored)	RW	0x0
14	TX_FSSLEW : TX_FSSLEW=0: Low speed slew rate TX_FSSLEW=1: Full speed slew rate	RW	0x0
13	TX_PD : TX power down override (if override enable is set). 1 = powered down.	RW	0x0
12	RX_PD : RX power down override (if override enable is set). 1 = powered down.	RW	0x0
11	TX_DM : Output data. TX_DIFFMODE=1, Ignored TX_DIFFMODE=0, Drives DPM only. TX_DM_OE=1 to enable drive. DPM=TX_DM	RW	0x0
10	TX_DP : Output data. If TX_DIFFMODE=1, Drives DPP/DPM diff pair. TX_DP_OE=1 to enable drive. DPP=TX_DP, DPM=~TX_DP If TX_DIFFMODE=0, Drives DPP only. TX_DP_OE=1 to enable drive. DPP=TX_DP	RW	0x0
9	TX_DM_OE : Output enable. If TX_DIFFMODE=1, Ignored. If TX_DIFFMODE=0, OE for DPM only. 0 - DPM in Hi-Z state; 1 - DPM driving	RW	0x0
8	TX_DP_OE : Output enable. If TX_DIFFMODE=1, OE for DPP/DPM diff pair. 0 - DPP/DPM in Hi-Z state; 1 - DPP/DPM driving If TX_DIFFMODE=0, OE for DPP only. 0 - DPP in Hi-Z state; 1 - DPP driving	RW	0x0
7	Reserved.	-	-

位	描述	类型	复位值
5	OVERCURR_DETECT_EN	读写	0x0
4	OVERCURR_DETECT	读写	0x0
3	VBUS_DETECT_OVERRIDE_EN	读写	0x0
2	VBUS_DETECT	读写	0x0
1	VBUS_EN_OVERRIDE_EN	读写	0x0
0	VBUS_EN	读写	0x0

USB: USBPHY_DIRECT 寄存器

偏移: 0x7C

描述

此寄存器允许直接控制 USB 物理层。需配合 usbphy_direct_override 寄存器使用以启用各覆盖位。

表 415。
USBPHY_DIRECT
寄存器

位	描述	类型	复位值
31:23	保留。	-	-
22	DM_OVV : DM 过压	只读	0x0
21	DP_OVV : DP 过压	只读	0x0
20	DM_OVCN : DM 过流	只读	0x0
19	DP_OVCN : DP 过流	只读	0x0
18	RX_DM : DPM 引脚状态	只读	0x0
17	RX_DP : DPP 引脚状态	只读	0x0
16	RX_DD : 差分接收	只读	0x0
15	TX_DIFFMODE : TX_DIFFMODE=0: 单端模式 TX_DIFFMODE=1: 差分驱动模式 (忽略 TX_DM 和 TX_DM_OE)	读写	0x0
14	TX_FSSLEW : TX_FSSLEW=0: 低速转换速率 TX_FSSLEW=1: 全速转换速率	读写	0x0
13	TX_PD : TX 电源关闭覆盖 (如启用覆盖)。1 = 电源关闭。	读写	0x0
12	RX_PD : RX 电源关闭覆盖 (如启用覆盖)。1 = 电源关闭。	读写	0x0
11	TX_DM : 输出数据。TX_DIFFMODE=1 时忽略。 TX_DIFFMODE=0 时, 仅驱动 DPM。TX_DM_OE=1 用于启用驱动。 DPM=TX_DM	读写	0x0
10	TX_DP : 输出数据。当 TX_DIFFMODE=1 时, 驱动 DPP/DPM 差分对。 TX_DP_OE=1 用于启用驱动。DPP=TX_DP, DPM=~TX_DP 如果 TX_DIFFMODE=0, 仅驱动 DPP。TX_DP_OE=1 以启用驱动。 DPP=TX_DP	读写	0x0
9	TX_DM_OE : 输出使能。如果 TX_DIFFMODE=1, 则忽略。 如果 TX_DIFFMODE=0, 仅对 DPM 使能 OE。0 - DPM 处于高阻态; 1 - DPM 驱动中。	读写	0x0
8	TX_DP_OE : 输出使能。如果 TX_DIFFMODE=1, 对 DPP/DPM 差分对使能 OE。 0 - DPP/DPM 处于高阻态; 1 - DPP/DPM 驱动中。 如果 TX_DIFFMODE=0, 仅对 DPP 使能 OE。0 - DPP 处于高阻态; 1 - DPP 驱动中。	读写	0x0
7	保留。	-	-

Bits	Description	Type	Reset
6	DM_PULLDN_EN: DM pull down enable	RW	0x0
5	DM_PULLUP_EN: DM pull up enable	RW	0x0
4	DM_PULLUP_HISEL: Enable the second DM pull up resistor. 0 - Pull = Rpu2; 1 - Pull = Rpu1 + Rpu2	RW	0x0
3	Reserved.	-	-
2	DP_PULLDN_EN: DP pull down enable	RW	0x0
1	DP_PULLUP_EN: DP pull up enable	RW	0x0
0	DP_PULLUP_HISEL: Enable the second DP pull up resistor. 0 - Pull = Rpu2; 1 - Pull = Rpu1 + Rpu2	RW	0x0

USB: USBPHY_DIRECT_OVERRIDE Register

Offset: 0x80

Description

Override enable for each control in usbphy_direct

Table 416.
USBPHY_DIRECT_OVERRIDE Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15	TX_DIFFMODE_OVERRIDE_EN	RW	0x0
14:13	Reserved.	-	-
12	DM_PULLUP_OVERRIDE_EN	RW	0x0
11	TX_FSSLEW_OVERRIDE_EN	RW	0x0
10	TX_PD_OVERRIDE_EN	RW	0x0
9	RX_PD_OVERRIDE_EN	RW	0x0
8	TX_DM_OVERRIDE_EN	RW	0x0
7	TX_DP_OVERRIDE_EN	RW	0x0
6	TX_DM_OE_OVERRIDE_EN	RW	0x0
5	TX_DP_OE_OVERRIDE_EN	RW	0x0
4	DM_PULLDN_EN_OVERRIDE_EN	RW	0x0
3	DP_PULLDN_EN_OVERRIDE_EN	RW	0x0
2	DP_PULLUP_EN_OVERRIDE_EN	RW	0x0
1	DM_PULLUP_HISEL_OVERRIDE_EN	RW	0x0
0	DP_PULLUP_HISEL_OVERRIDE_EN	RW	0x0

USB: USBPHY_TRIM Register

Offset: 0x84

Description

Used to adjust trim values of USB phy pull down resistors.

位	描述	类型	复位值
6	DM_PULLDN_EN : DM 下拉使能。	读写	0x0
5	DM_PULLUP_EN : DM 上拉使能。	读写	0x0
4	DM_PULLUP_HISEL : 启用第二个 DM 上拉电阻。0 - 上拉为 Rpu2; 1 - 上拉为 Rpu1 + Rpu2。	读写	0x0
3	保留。	-	-
2	DP_PULLDN_EN : DP 下拉使能	读写	0x0
1	DP_PULLUP_EN : DP 上拉使能	读写	0x0
0	DP_PULLUP_HISEL : 启用第二个 DP 上拉电阻。0 - 拉低 = Rpu2; 1 - 拉低 = Rpu1 + Rpu2	读写	0x0

USB: USBPHY_DIRECT_OVERRIDE 寄存器

偏移: 0x80

描述

usbphy_direct 中各控制项的覆盖使能。

表 416。
USBPHY_DIRECT_OVERRIDE 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	TX_DIFFMODE_OVERRIDE_EN	读写	0x0
14:13	保留。	-	-
12	DM_PULLUP_OVERRIDE_EN	读写	0x0
11	TX_FSSLEW_OVERRIDE_EN	读写	0x0
10	TX_PD_OVERRIDE_EN	读写	0x0
9	RX_PD_OVERRIDE_EN	读写	0x0
8	TX_DM_OVERRIDE_EN	读写	0x0
7	TX_DP_OVERRIDE_EN	读写	0x0
6	TX_DM_OE_OVERRIDE_EN	读写	0x0
5	TX_DP_OE_OVERRIDE_EN	读写	0x0
4	DM_PULLDN_EN_OVERRIDE_EN	读写	0x0
3	DP_PULLDN_EN_OVERRIDE_EN	读写	0x0
2	DP_PULLUP_EN_OVERRIDE_EN	读写	0x0
1	DM_PULLUP_HISEL_OVERRIDE_EN	读写	0x0
0	DP_PULLUP_HISEL_OVERRIDE_EN	读写	0x0

USB: USBPHY_TRIM 寄存器

偏移: 0x84

描述

用于调整 USB 物理层下拉电阻的调节值。

Table 417.
USBPHY_TRIM
Register

Bits	Description	Type	Reset
31:13	Reserved.	-	-
12:8	DM_PULLDN_TRIM: Value to drive to USB PHY DM pulldown resistor trim control Experimental data suggests that the reset value will work, but this register allows adjustment if required	RW	0x1f
7:5	Reserved.	-	-
4:0	DP_PULLDN_TRIM: Value to drive to USB PHY DP pulldown resistor trim control Experimental data suggests that the reset value will work, but this register allows adjustment if required	RW	0x1f

USB: INTR Register

Offset: 0x8c

Description

Raw Interrupts

Table 418. INTR
Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19	EP_STALL_NAK: Raised when any bit in EP_STATUS_STALL_NAK is set. Clear by clearing all bits in EP_STATUS_STALL_NAK.	RO	0x0
18	ABORT_DONE: Raised when any bit in ABORT_DONE is set. Clear by clearing all bits in ABORT_DONE.	RO	0x0
17	DEV_SOF: Set every time the device receives a SOF (Start of Frame) packet. Cleared by reading SOF_RD	RO	0x0
16	SETUP_REQ: Device. Source: SIE_STATUS.SETUP_REC	RO	0x0
15	DEV_RESUME_FROM_HOST: Set when the device receives a resume from the host. Cleared by writing to SIE_STATUS.RESUME	RO	0x0
14	DEV_SUSPEND: Set when the device suspend state changes. Cleared by writing to SIE_STATUS.SUSPENDED	RO	0x0
13	DEV_CONN_DIS: Set when the device connection state changes. Cleared by writing to SIE_STATUS.CONNECTED	RO	0x0
12	BUS_RESET: Source: SIE_STATUS.BUS_RESET	RO	0x0
11	VBUS_DETECT: Source: SIE_STATUS.VBUS_DETECTED	RO	0x0
10	STALL: Source: SIE_STATUS.STALL_REC	RO	0x0
9	ERROR_CRC: Source: SIE_STATUS.CRC_ERROR	RO	0x0
8	ERROR_BIT_STUFF: Source: SIE_STATUS.BIT_STUFF_ERROR	RO	0x0
7	ERROR_RX_OVERFLOW: Source: SIE_STATUS.RX_OVERFLOW	RO	0x0
6	ERROR_RX_TIMEOUT: Source: SIE_STATUS.RX_TIMEOUT	RO	0x0
5	ERROR_DATA_SEQ: Source: SIE_STATUS.DATA_SEQ_ERROR	RO	0x0
4	BUFF_STATUS: Raised when any bit in BUFF_STATUS is set. Clear by clearing all bits in BUFF_STATUS.	RO	0x0

表 417。
USBPHY_TRI
M 寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12:8	DM_PULLDN_TRIM : 驱动至 USB PHY 的数值 DM 下拉电阻微调控制 实验数据表明复位值有效，但此寄存器允许根据需要进行调整	读写	0x1f
7:5	保留。	-	-
4:0	DP_PULLDN_TRIM : 驱动至 USB PHY 的数值 DP 下拉电阻微调控制 实验数据表明复位值有效，但此寄存器允许根据需要进行调整	读写	0x1f

USB：INTR 寄存器

偏移：0x8c

描述

原始中断

表 418. INTR
寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	只读	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	只读	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	只读	0x0
16	SETUP_REQ : 设备。来源：SIE_STATUS.SETUP_REC	只读	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	只读	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	只读	0x0
12	BUS_RESET : 来源：SIE_STATUS.BUS_RESET	只读	0x0
11	VBUS_DETECT : 来源：SIE_STATUS.VBUS_DETECTED	只读	0x0
10	STALL : 来源：SIE_STATUS.STALL_REC	只读	0x0
9	ERROR_CRC : 来源：SIE_STATUS.CRC_ERROR	只读	0x0
8	ERROR_BIT_STUFF : 来源：SIE_STATUS.BIT_STUFF_ERROR	只读	0x0
7	ERROR_RX_OVERFLOW : 来源：SIE_STATUS.RX_OVERFLOW	只读	0x0
6	ERROR_RX_TIMEOUT : 来源：SIE_STATUS.RX_TIMEOUT	只读	0x0
5	ERROR_DATA_SEQ : 来源：SIE_STATUS.DATA_SEQ_ERROR	只读	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	只读	0x0

Bits	Description	Type	Reset
3	TRANS_COMPLETE : Raised every time SIE_STATUS.TRANS_COMPLETE is set. Clear by writing to this bit.	RO	0x0
2	HOST_SOF : Host: raised every time the host sends a SOF (Start of Frame). Cleared by reading SOF_RD	RO	0x0
1	HOST_RESUME : Host: raised when a device wakes up the host. Cleared by writing to SIE_STATUS.RESUME	RO	0x0
0	HOST_CONN_DIS : Host: raised when a device is connected or disconnected (i.e. when SIE_STATUS.SPEED changes). Cleared by writing to SIE_STATUS.SPEED	RO	0x0

USB: INTE Register

Offset: 0x90

Description

Interrupt Enable

Table 419. INTE Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19	EP_STALL_NAK : Raised when any bit in EP_STATUS_STALL_NAK is set. Clear by clearing all bits in EP_STATUS_STALL_NAK.	RW	0x0
18	ABORT_DONE : Raised when any bit in ABORT_DONE is set. Clear by clearing all bits in ABORT_DONE.	RW	0x0
17	DEV_SOF : Set every time the device receives a SOF (Start of Frame) packet. Cleared by reading SOF_RD	RW	0x0
16	SETUP_REQ : Device. Source: SIE_STATUS.SETUP_REC	RW	0x0
15	DEV_RESUME_FROM_HOST : Set when the device receives a resume from the host. Cleared by writing to SIE_STATUS.RESUME	RW	0x0
14	DEV_SUSPEND : Set when the device suspend state changes. Cleared by writing to SIE_STATUS.SUSPENDED	RW	0x0
13	DEV_CONN_DIS : Set when the device connection state changes. Cleared by writing to SIE_STATUS.CONNECTED	RW	0x0
12	BUS_RESET : Source: SIE_STATUS.BUS_RESET	RW	0x0
11	VBUS_DETECT : Source: SIE_STATUS.VBUS_DETECTED	RW	0x0
10	STALL : Source: SIE_STATUS.STALL_REC	RW	0x0
9	ERROR_CRC : Source: SIE_STATUS.CRC_ERROR	RW	0x0
8	ERROR_BIT_STUFF : Source: SIE_STATUS.BIT_STUFF_ERROR	RW	0x0
7	ERROR_RX_OVERFLOW : Source: SIE_STATUS.RX_OVERFLOW	RW	0x0
6	ERROR_RX_TIMEOUT : Source: SIE_STATUS.RX_TIMEOUT	RW	0x0
5	ERROR_DATA_SEQ : Source: SIE_STATUS.DATA_SEQ_ERROR	RW	0x0
4	BUFF_STATUS : Raised when any bit in BUFF_STATUS is set. Clear by clearing all bits in BUFF_STATUS.	RW	0x0
3	TRANS_COMPLETE : Raised every time SIE_STATUS.TRANS_COMPLETE is set. Clear by writing to this bit.	RW	0x0

位	描述	类型	复位值
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	只读	0x0
2	HOST_SOF : 主机：每次主机发送 SOF（帧起始）时触发。 通过读取 SOF_RD 清除	只读	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	只读	0x0

USB: INTE 寄存器

偏移: 0x90

描述

中断使能

表 419. INTE
寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	读写	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	读写	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	读写	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	读写	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	读写	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	读写	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	读写	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	读写	0x0
10	STALL : 来源: SIE_STATUS.STALL_REC	读写	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	读写	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	读写	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	读写	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	读写	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	读写	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	读写	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	读写	0x0

Bits	Description	Type	Reset
2	HOST_SOF : Host: raised every time the host sends a SOF (Start of Frame). Cleared by reading SOF_RD	RW	0x0
1	HOST_RESUME : Host: raised when a device wakes up the host. Cleared by writing to SIE_STATUS.RESUME	RW	0x0
0	HOST_CONN_DIS : Host: raised when a device is connected or disconnected (i.e. when SIE_STATUS.SPEED changes). Cleared by writing to SIE_STATUS.SPEED	RW	0x0

USB: INTF Register

Offset: 0x94

Description

Interrupt Force

Table 420. INTF Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19	EP_STALL_NAK : Raised when any bit in EP_STATUS_STALL_NAK is set. Clear by clearing all bits in EP_STATUS_STALL_NAK.	RW	0x0
18	ABORT_DONE : Raised when any bit in ABORT_DONE is set. Clear by clearing all bits in ABORT_DONE.	RW	0x0
17	DEV_SOF : Set every time the device receives a SOF (Start of Frame) packet. Cleared by reading SOF_RD	RW	0x0
16	SETUP_REQ : Device. Source: SIE_STATUS.SETUP_REC	RW	0x0
15	DEV_RESUME_FROM_HOST : Set when the device receives a resume from the host. Cleared by writing to SIE_STATUS.RESUME	RW	0x0
14	DEV_SUSPEND : Set when the device suspend state changes. Cleared by writing to SIE_STATUS.SUSPENDED	RW	0x0
13	DEV_CONN_DIS : Set when the device connection state changes. Cleared by writing to SIE_STATUS.CONNECTED	RW	0x0
12	BUS_RESET : Source: SIE_STATUS.BUS_RESET	RW	0x0
11	VBUS_DETECT : Source: SIE_STATUS.VBUS_DETECTED	RW	0x0
10	STALL : Source: SIE_STATUSSTALL_REC	RW	0x0
9	ERROR_CRC : Source: SIE_STATUS.CRC_ERROR	RW	0x0
8	ERROR_BIT_STUFF : Source: SIE_STATUS.BIT_STUFF_ERROR	RW	0x0
7	ERROR_RX_OVERFLOW : Source: SIE_STATUS.RX_OVERFLOW	RW	0x0
6	ERROR_RX_TIMEOUT : Source: SIE_STATUS.RX_TIMEOUT	RW	0x0
5	ERROR_DATA_SEQ : Source: SIE_STATUS.DATA_SEQ_ERROR	RW	0x0
4	BUFF_STATUS : Raised when any bit in BUFF_STATUS is set. Clear by clearing all bits in BUFF_STATUS.	RW	0x0
3	TRANS_COMPLETE : Raised every time SIE_STATUS.TRANS_COMPLETE is set. Clear by writing to this bit.	RW	0x0
2	HOST_SOF : Host: raised every time the host sends a SOF (Start of Frame). Cleared by reading SOF_RD	RW	0x0

位	描述	类型	复位值
2	HOST_SOF : 主机：每次主机发送 SOF（帧起始）时触发。 通过读取 SOF_RD 清除	读写	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	读写	0x0

USB: INTF 寄存器

偏移: 0x94

描述

中断强制

表 420. INTF 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	读写	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	读写	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	读写	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	读写	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	读写	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	读写	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	读写	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	读写	0x0
10	STALL : 来源: SIE_STATUSSTALL_REC	读写	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	读写	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	读写	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	读写	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	读写	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	读写	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	读写	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	读写	0x0
2	HOST_SOF : 主机：每次主机发送 SOF（帧起始）时触发。 通过读取 SOF_RD 清除	读写	0x0

Bits	Description	Type	Reset
1	HOST_RESUME : Host: raised when a device wakes up the host. Cleared by writing to SIE_STATUS.RESUME	RW	0x0
0	HOST_CONN_DIS : Host: raised when a device is connected or disconnected (i.e. when SIE_STATUS.SPEED changes). Cleared by writing to SIE_STATUS.SPEED	RW	0x0

USB: INTS Register

Offset: 0x98

Description

Interrupt status after masking & forcing

Table 421. INTS Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19	EP_STALL_NAK : Raised when any bit in EP_STATUS_STALL_NAK is set. Clear by clearing all bits in EP_STATUS_STALL_NAK.	RO	0x0
18	ABORT_DONE : Raised when any bit in ABORT_DONE is set. Clear by clearing all bits in ABORT_DONE.	RO	0x0
17	DEV_SOF : Set every time the device receives a SOF (Start of Frame) packet. Cleared by reading SOF_RD	RO	0x0
16	SETUP_REQ : Device. Source: SIE_STATUS.SETUP_REC	RO	0x0
15	DEV_RESUME_FROM_HOST : Set when the device receives a resume from the host. Cleared by writing to SIE_STATUS.RESUME	RO	0x0
14	DEV_SUSPEND : Set when the device suspend state changes. Cleared by writing to SIE_STATUS.SUSPENDED	RO	0x0
13	DEV_CONN_DIS : Set when the device connection state changes. Cleared by writing to SIE_STATUS.CONNECTED	RO	0x0
12	BUS_RESET : Source: SIE_STATUS.BUS_RESET	RO	0x0
11	VBUS_DETECT : Source: SIE_STATUS.VBUS_DETECTED	RO	0x0
10	STALL : Source: SIE_STATUSSTALL_REC	RO	0x0
9	ERROR_CRC : Source: SIE_STATUS.CRC_ERROR	RO	0x0
8	ERROR_BIT_STUFF : Source: SIE_STATUS.BIT_STUFF_ERROR	RO	0x0
7	ERROR_RX_OVERFLOW : Source: SIE_STATUS.RX_OVERFLOW	RO	0x0
6	ERROR_RX_TIMEOUT : Source: SIE_STATUS.RX_TIMEOUT	RO	0x0
5	ERROR_DATA_SEQ : Source: SIE_STATUS.DATA_SEQ_ERROR	RO	0x0
4	BUFF_STATUS : Raised when any bit in BUFF_STATUS is set. Clear by clearing all bits in BUFF_STATUS.	RO	0x0
3	TRANS_COMPLETE : Raised every time SIE_STATUS.TRANS_COMPLETE is set. Clear by writing to this bit.	RO	0x0
2	HOST_SOF : Host: raised every time the host sends a SOF (Start of Frame). Cleared by reading SOF_RD	RO	0x0
1	HOST_RESUME : Host: raised when a device wakes up the host. Cleared by writing to SIE_STATUS.RESUME	RO	0x0

位	描述	类型	复位值
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	读写	0x0
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	读写	0x0

USB: INTS 寄存器

偏移: 0x98

说明

掩码及强制后的中断状态

表 421. INTS 寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19	EP_STALL_NAK : 当 EP_STATUS_STALL_NAK 中任一位被置位时触发。通过清除 EP_STATUS_STALL_NAK 中所有位清除该状态。	只读	0x0
18	ABORT_DONE : 当 ABORT_DONE 中任一位被置位时触发。通过清除 ABORT_DONE 中所有位清除该状态。	只读	0x0
17	DEV_SOF : 设备每接收一帧起始 (SOF, Start of Frame) 包时置位。 通过读取 SOF_RD 清除	只读	0x0
16	SETUP_REQ : 设备。来源: SIE_STATUS.SETUP_REC	只读	0x0
15	DEV_RESUME_FROM_HOST : 当设备接收到主机恢复信号时置位。通过写入 SIE_STATUS.RESUME 清除	只读	0x0
14	DEV_SUSPEND : 当设备挂起状态变更时置位。通过写入 SIE_STATUS.SUSPENDED 清除	只读	0x0
13	DEV_CONN_DIS : 当设备连接状态变更时置位。通过写入 SIE_STATUS.CONNECTED 清除	只读	0x0
12	BUS_RESET : 来源: SIE_STATUS.BUS_RESET	只读	0x0
11	VBUS_DETECT : 来源: SIE_STATUS.VBUS_DETECTED	只读	0x0
10	STALL : 来源: SIE_STATUSSTALL_REC	只读	0x0
9	ERROR_CRC : 来源: SIE_STATUS.CRC_ERROR	只读	0x0
8	ERROR_BIT_STUFF : 来源: SIE_STATUS.BIT_STUFF_ERROR	只读	0x0
7	ERROR_RX_OVERFLOW : 来源: SIE_STATUS.RX_OVERFLOW	只读	0x0
6	ERROR_RX_TIMEOUT : 来源: SIE_STATUS.RX_TIMEOUT	只读	0x0
5	ERROR_DATA_SEQ : 来源: SIE_STATUS.DATA_SEQ_ERROR	只读	0x0
4	BUFF_STATUS : 当 BUFF_STATUS 任意位被置位时触发。通过清除 BUFF_STATUS 内所有位来复位。	只读	0x0
3	TRANS_COMPLETE : 每次 SIE_STATUS.TRANS_COMPLETE 置位时触发。 通过写入该位来复位。	只读	0x0
2	HOST_SOF : 主机：每次主机发送 SOF (帧起始) 时触发。 通过读取 SOF_RD 清除	只读	0x0
1	HOST_RESUME : 主机：设备唤醒主机时触发。通过写入 SIE_STATUS.RESUME 清除	只读	0x0

Bits	Description	Type	Reset
0	HOST_CONN_DIS : Host: raised when a device is connected or disconnected (i.e. when SIE_STATUS.SPEED changes). Cleared by writing to SIE_STATUS.SPEED	RO	0x0

References

- [1] [USB Made Simple](#)
- [2] [USB 2.0 Specification](#)

4.2. UART

ARM Documentation

Excerpted from the [PrimeCell UART \(PL011\) Technical Reference Manual](#). Used with permission.

RP2040 has 2 identical instances of a UART peripheral, based on the ARM Primecell UART (PL011) (Revision r1p5).

Each instance supports the following features:

- Separate 32x8 Tx and 32x12 Rx FIFOs
- Programmable baud rate generator, clocked by `clk_peri` (see [Section 2.15.1](#))
- Standard asynchronous communication bits (start, stop, parity) added on transmit and removed on receive
- line break detection
- programmable serial interface (5, 6, 7, or 8 bits)
- 1 or 2 stop bits
- programmable hardware flow control

Each UART can be connected to a number of GPIO pins as defined in the GPIO muxing table in [Section 2.19.2](#). Connections to the GPIO muxing are prefixed with the UART instance name `uart0_` or `uart1_`, and include the following:

- Transmit data `tx` (referred to as UARTRXD in the following sections)
- Received data `rx` (referred to as UARTTXD in the following sections)
- Output flow control `rts` (referred to as nUARTRTS in the following sections)
- Input flow control `cts` (referred to as nUARTCTS in the following sections)

The modem mode and IrDA mode of the PL011 are not supported.

The `UARTCLK` is driven from `clk_peri`, and `PCLK` is driven from the system clock `clk_sys` (see [Section 2.15.1](#)).

4.2.1. Overview

The UART performs:

- Serial-to-parallel conversion on data received from a peripheral device
- Parallel-to-serial conversion on data transmitted to the peripheral device.

The CPU reads and writes data and control/status information through the AMBA APB interface. The transmit and receive paths are buffered with internal FIFO memories enabling up to 32-bytes to be stored independently in both

位	描述	类型	复位值
0	HOST_CONN_DIS : 主机：设备连接或断开（即 SIE_STATUS.SPEED 变化）时触发。通过写入 SIE_STATUS.SPEED 清除	只读	0x0

参考文献

- [1] USB 简明教程
- [2] USB 2.0 规范

4.2. UART

ARM文档

节选自 PrimeCell UART (PL011) 技术参考手册，已获授权使用。

RP2040 拥有两个基于 ARM Primecell UART (PL011) (修订版 r1p5) 的同型号 UART 外设实例。

每个实例支持以下功能：

- 独立的 32×8 发送和 32×12 接收 FIFO 缓冲区
- 可编程波特率发生器，由 `clk_peri` 时钟驱动（详见第 2.15.1 节）
- 发送时添加且接收时去除的标准异步通信位（起始位、停止位、校验位）
- 换行符检测
- 可编程串行接口（5、6、7 或 8 位）
- 1 或 2 个停止位
- 可编程硬件流控制

每个UART可连接至多个GPIO引脚，详见第2.19.2节中的GPIO复用表。

GPIO复用连接前缀为UART实例名 `uart0_` 或 `uart1_`，包含以下内容：

- 发送数据 `tx`（以下章节称为UARTTXD）
- 接收数据 `rx`（以下章节称为UARTRXD）
- 输出流控制 `rts`（以下章节称为nUARTRTS）
- 输入流控制 `cts`（以下章节称为nUARTCTS）。PL011的调制解调器模式及

IrDA模式不支持。

UART`CLK`由`clk_peri`驱动，`PCLK`由系统时钟 `clk_sys`驱动（详见第2.15.1节）。

4.2.1. 概述

UART执行以下操作：

- 对从外围设备接收的数据进行串行转并行转换。
- 对发送至外围设备的数据进行并行转串行转换。

CPU通过AMBA APB接口读写数据及控制/状态信息。发送和接收路径均通过内部FIFO缓存独立缓冲，可分别存储最多32字节的数据，

transmit and receive modes.

The UART:

- Includes a programmable baud rate generator that generates a common transmit and receive internal clock from the UART internal reference clock input, UARTCLK
- Offers similar functionality to the industry-standard 16C650 UART device
- Supports a maximum baud rate of UARTCLK / 16 in UART mode (7.8 Mbaud at 125MHz)

The UART operation and baud rate values are controlled by the Line Control Register, [UARTLCR_H](#) and the baud rate divisor registers (Integer Baud Rate Register, [UARTIBRD](#) and Fractional Baud Rate Register, [UARTFBRD](#)).

The UART can generate:

- Individually-maskable interrupts from the receive (including timeout), transmit, modem status and error conditions
- A single combined interrupt so that the output is asserted if any of the individual interrupts are asserted, and unmasked
- DMA request signals for interfacing with a Direct Memory Access (DMA) controller.

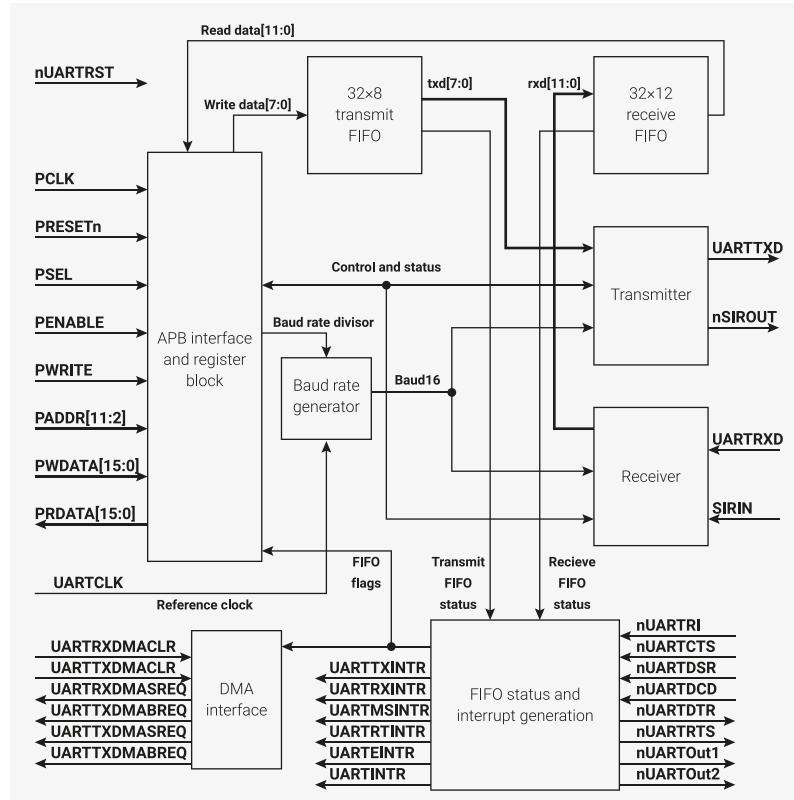
If a framing, parity, or break error occurs during reception, the appropriate error bit is set, and is stored in the FIFO. If an overrun condition occurs, the overrun register bit is set immediately and FIFO data is prevented from being overwritten.

You can program the FIFOs to be 1-byte deep providing a conventional double-buffered UART interface.

There is a programmable hardware flow control feature that uses the nUARTCTS input and the nUARTRTS output to automatically control the serial data flow.

4.2.2. Functional description

Figure 59. UART block diagram. Test logic is not shown for clarity.



适用于发送与接收两种模式。

UART具备以下功能：

- 包含可编程波特率发生器，根据UART内部参考时钟输入UARTCLK生成统一的发送和接收内部时钟。
- 功能与行业标准16C650 UART设备相似。
- 在UART模式下支持最高波特率为UARTCLK / 16（125MHz时为7.8 Mbaud）。

UART的操作和波特率由线路控制寄存器UARTLCR_H及波特率除数寄存器（整数波特率寄存器UARTIBRD与小数波特率寄存器UARTFBRD）进行控制。

UART 可产生：

- 来自接收（包括超时）、发送、调制解调器状态及错误条件的可单独屏蔽中断
- 单个合并中断，若任一单个中断被触发，该输出即被置位，且未屏蔽
- 用于与直接存储器访问（DMA）控制器接口的 DMA 请求信号。

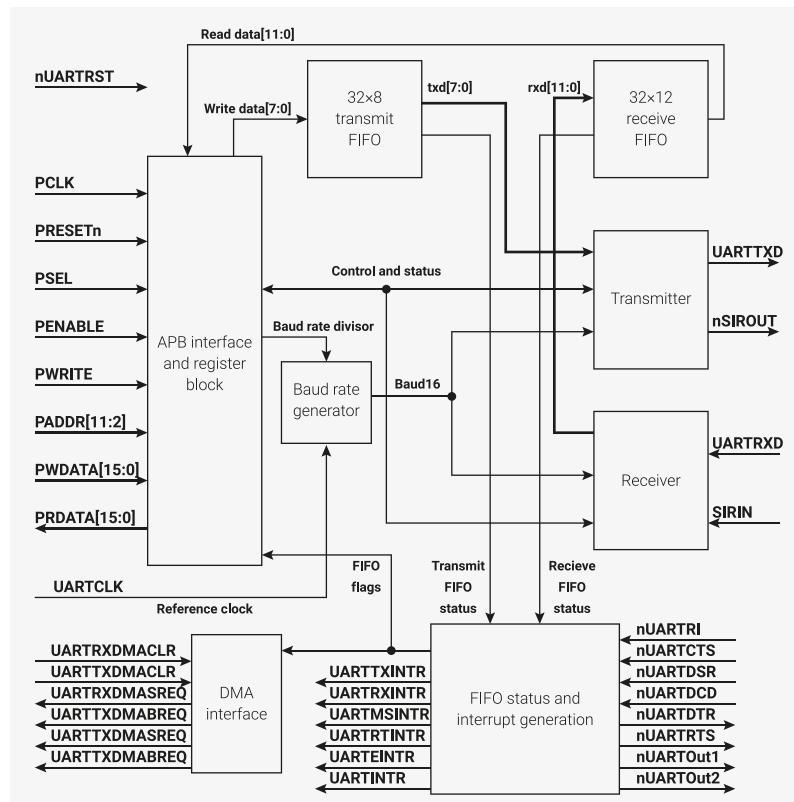
若接收过程中发生帧错误、奇偶校验错误或断续错误，相关错误位会被置位，并存储于 FIFO 中。若发生溢出条件，溢出寄存器位会即时置位，并防止 FIFO 数据被覆盖。

您可以将 FIFO 设置为 1 字节深度，以提供传统的双缓冲 UART 接口。

具备可编程硬件流控制功能，采用 nUARTCTS 输入和 nUARTRTS 输出自动控制串行数据流。

4.2.2. 功能描述

图 59. UART 区块图
。为保持清晰，未显示测试逻辑。



4.2.2.1. AMBA APB interface

The AMBA APB interface generates read and write decodes for accesses to status/control registers, and the transmit and receive FIFOs.

4.2.2.2. Register block

The register block stores data written, or to be read across the AMBA APB interface.

4.2.2.3. Baud rate generator

The baud rate generator contains free-running counters that generate the internal clocks: Baud16 and IrLPBaud16 signals. Baud16 provides timing information for UART transmit and receive control. Baud16 is a stream of pulses with a width of one UARTCLK clock period and a frequency of 16 times the baud rate.

4.2.2.4. Transmit FIFO

The transmit FIFO is an 8-bit wide, 32 location deep, FIFO memory buffer. CPU data written across the APB interface is stored in the FIFO until read out by the transmit logic. You can disable the transmit FIFO to act like a one-byte holding register.

4.2.2.5. Receive FIFO

The receive FIFO is a 12-bit wide, 32 location deep, FIFO memory buffer. Received data and corresponding error bits, are stored in the receive FIFO by the receive logic until read out by the CPU across the APB interface. The receive FIFO can be disabled to act like a one-byte holding register.

4.2.2.6. Transmit logic

The transmit logic performs parallel-to-serial conversion on the data read from the transmit FIFO. Control logic outputs the serial bit stream beginning with a start bit, data bits with the Least Significant Bit (LSB) first, followed by the parity bit, and then the stop bits according to the programmed configuration in control registers.

4.2.2.7. Receive logic

The receive logic performs serial-to-parallel conversion on the received bit stream after a valid start pulse has been detected. Overrun, parity, frame error checking, and line break detection are also performed, and their status accompanies the data that is written to the receive FIFO.

4.2.2.8. Interrupt generation logic

Individual maskable active HIGH interrupts are generated by the UART. A combined interrupt output is generated as an OR function of the individual interrupt requests and is connected to the processor interrupt controllers.

See [Section 4.2.6](#) for more information.

4.2.2.1. AMBA APB 接口

AMBA APB接口产生用于访问状态/控制寄存器及发送和接收FIFO的读写译码。

4.2.2.2. 寄存器块

寄存器块用于存储通过AMBA APB接口写入或将要读取的数据。

4.2.2.3. 波特率发生器

波特率发生器包含自由运行计数器，用于生成内部时钟信号：Baud16和IrLPBaud16。Baud16为UART发送和接收控制提供时序信息。Baud16为一系列脉冲，脉冲宽度为一个UARTCLK时钟周期，频率为波特率的16倍。

4.2.2.4. 发送FIFO

发送FIFO为8位宽、深度为32的位置的FIFO存储缓冲区。通过APB接口写入的CPU数据存储在FIFO中，直至由发送逻辑读取。您可以禁用发送FIFO，使其行为类似于单字节保持寄存器。

4.2.2.5. 接收FIFO

接收FIFO是一个12位宽、深度为32个单元的FIFO存储缓冲区。接收的数据及相应的错误位由接收逻辑存储于接收FIFO中，直至通过APB接口由CPU读取。接收FIFO可被禁用，以作为单字节保持寄存器使用。

4.2.2.6. 发送逻辑

发送逻辑对从发送FIFO读取的数据执行并行转串行转换。控制逻辑依照控制寄存器中设定的配置，输出从起始位开始的串行比特流，数据位最低有效位（LSB）先行，随后为奇偶校验位和停止位。

4.2.2.7. 接收逻辑

接收逻辑在检测到有效的起始脉冲后，对接收的比特流执行串行转并行转换。还会执行溢出、奇偶校验、帧错误检测及线路中断检测，其状态信息随数据一并写入接收FIFO。

4.2.2.8. 中断生成逻辑

UART会产生单独的高电平有效可屏蔽中断。各单独中断请求的逻辑或（OR）功能生成组合中断输出，并连接至处理器中断控制器。

详见第 4.2.6 节。

4.2.2.9. DMA interface

The UART provides an interface to connect to the DMA controller as UART DMA interface in [Section 4.2.5](#) describes.

4.2.2.10. Synchronizing registers and logic

The UART supports both asynchronous and synchronous operation of the clocks, PCLK and UARTCLK. Synchronization registers and handshaking logic have been implemented, and are active at all times. This has a minimal impact on performance or area. Synchronization of control signals is performed on both directions of data flow, that is from the PCLK to the UARTCLK domain, and from the UARTCLK to the PCLK domain.

4.2.3. Operation

4.2.3.1. Clock signals

The frequency selected for UARTCLK must accommodate the required range of baud rates:

- FUARTCLK (min) $\geq 16 \times \text{baud_rate(max)}$
- FUARTCLK(max) $\leq 16 \times 65535 \times \text{baud_rate(min)}$

For example, for a range of baud rates from 110 baud to 460800 baud the UARTCLK frequency must be between 7.3728MHz to 115.34MHz.

The frequency of UARTCLK must also be within the required error limits for all baud rates to be used.

There is also a constraint on the ratio of clock frequencies for PCLK to UARTCLK. The frequency of UARTCLK must be no more than 5/3 times faster than the frequency of PCLK:

- FUARTCLK $\leq 5/3 \times \text{FPCLK}$

For example, in UART mode, to generate 921600 baud when UARTCLK is 14.7456MHz then PCLK must be greater than or equal to 8.85276MHz. This ensures that the UART has sufficient time to write the received data to the receive FIFO.

4.2.3.2. UART operation

Control data is written to the UART Line Control Register, UARTLCR. This register is 30-bits wide internally, but is externally accessed through the APB interface by writes to the following registers:

The [UARTLCR_H](#) register defines the:

- transmission parameters
- word length
- buffer mode
- number of transmitted stop bits
- parity mode
- break generation.

The [UARTIBRD](#) register defines the integer baud rate divider, and the [UARTFBRD](#) register defines the fractional baud rate divider.

4.2.2.9. DMA 接口

UART 提供连接至 DMA 控制器的接口，详细描述见第 4.2.5 节的 UART DMA 接口。

4.2.2.10. 同步寄存器与逻辑

UART 支持时钟 PCLK 与 UARTCLK 的异步及同步操作，已实现同步寄存器与握手机制，且始终保持激活。此设计对性能或芯片面积的影响极小。控制信号的同步在数据流双向方向进行，即从 PCLK 域到 UARTCLK 域，以及从 UARTCLK 域到 PCLK 域。

4.2.3. 操作

4.2.3.1. 时钟信号

为UARTCLK选择的频率必须满足所需的波特率范围：

- FUARTCLK（最小值） $\geq 16 \times \text{baud_rate}$ （最大值）
- FUARTCLK（最大值） $\leq 16 \times 65535 \times \text{baud_rate}$ （最小值）

例如，对于110波特到460800波特的波特率范围，UARTCLK频率必须在7.3728MHz至115.34MHz之间。

UARTCLK的频率还必须在所有所用波特率的规定误差范围内。

对于PCLK与UARTCLK的时钟频率比也有约束。UARTCLK频率不得超过PCLK频率的5/3倍：

- FUARTCLK $\leq \frac{5}{3} \times \text{FPCLK}$

例如，在UART模式下，若UARTCLK为14.7456MHz，要生成921600波特，PCLK必须大于或等于8.85276MHz，以确保UART有足够时间将接收数据写入接收FIFO。

4.2.3.2. UART 操作

控制数据写入 UART 线路控制寄存器 UARTLCR。该寄存器内部宽度为 30 位，但通过 APB 接口对以下寄存器进行写访问：

UARTLCR_H 寄存器定义：

- 传输参数
- 字长
- 缓冲模式
- 发送停止位数
- 奇偶校验模式
- 断线生成。

UARTIBRD 寄存器定义整数波特率分频器，UARTFBRD 寄存器定义小数波特率分频器。

4.2.3.2.1. Fractional baud rate divider

The baud rate divisor is a 22-bit number consisting of a 16-bit integer and a 6-bit fractional part. This is used by the baud rate generator to determine the bit period. The fractional baud rate divider enables the use of any clock with a frequency >3.6864MHz to act as UARTCLK, while it is still possible to generate all the standard baud rates.

The 16-bit integer is written to the Integer Baud Rate Register, [UARTIBRD](#). The 6-bit fractional part is written to the Fractional Baud Rate Register, [UARTFBRD](#). The Baud Rate Divisor has the following relationship to UARTCLK:

Baud Rate Divisor = $\text{UARTCLK}/(16 \times \text{Baud Rate}) = BRD_I + BRD_F$ where BRD_I is the integer part and BRD_F is the fractional part separated by a decimal point as [Figure 60](#).

Figure 60. Baud rate divisor.



You can calculate the 6-bit number (m) by taking the fractional part of the required baud rate divisor and multiplying it by 64 (that is, 2^n , where n is the width of the [UARTFBRD](#) Register) and adding 0.5 to account for rounding errors:

$$m = \text{integer}(BRD_F \times 2^n + 0.5)$$

An internal clock enable signal, Baud16, is generated, and is a stream of one UARTCLK wide pulses with an average frequency of 16 times the required baud rate. This signal is then divided by 16 to give the transmit clock. A low number in the baud rate divisor gives a short bit period, and a high number in the baud rate divisor gives a long bit period.

4.2.3.2.2. Data transmission or reception

Data received or transmitted is stored in two 32-byte FIFOs, though the receive FIFO has an extra four bits per character for status information. For transmission, data is written into the transmit FIFO. If the UART is enabled, it causes a data frame to start transmitting with the parameters indicated in the Line Control Register, [UARTLCR_H](#). Data continues to be transmitted until there is no data left in the transmit FIFO. The BUSY signal goes HIGH as soon as data is written to the transmit FIFO (that is, the FIFO is non-empty) and remains asserted HIGH while data is being transmitted. BUSY is negated only when the transmit FIFO is empty, and the last character has been transmitted from the shift register, including the stop bits. BUSY can be asserted HIGH even though the UART might no longer be enabled.

For each sample of data, three readings are taken and the majority value is kept. In the following paragraphs the middle sampling point is defined, and one sample is taken either side of it.

When the receiver is idle (UARTRXD continuously 1, in the marking state) and a LOW is detected on the data input (a start bit has been received), the receive counter, with the clock enabled by Baud16, begins running and data is sampled on the eighth cycle of that counter in UART mode, or the fourth cycle of the counter in SIR mode to allow for the shorter logic 0 pulses (half way through a bit period).

The start bit is valid if UARTRXD is still LOW on the eighth cycle of Baud16, otherwise a false start bit is detected and it is ignored.

If the start bit was valid, successive data bits are sampled on every 16th cycle of Baud16 (that is, one bit period later) according to the programmed length of the data characters. The parity bit is then checked if parity mode was enabled.

Lastly, a valid stop bit is confirmed if UARTRXD is HIGH, otherwise a framing error has occurred. When a full word is received, the data is stored in the receive FIFO, with any error bits associated with that word

4.2.3.2.3. Error bits

Three error bits are stored in bits [10:8] of the receive FIFO, and are associated with a particular character. There is an additional error that indicates an overrun error and this is stored in bit 11 of the receive FIFO.

4.2.3.2.1. 小数波特率分频器

波特率分频值为一个 22 位数字，由 16 位整数部分和 6 位小数部分组成。该值用于波特率发生器确定比特周期。小数波特率分频器允许使用任何频率大于 3.6864MHz 的时钟作为 UARTCLK，同时仍能生成所有标准波特率。

16 位整数写入整数波特率寄存器 UARTIBRD。6 位小数部分写入小数波特率寄存器 UARTFBRD。波特率除数与 UARTCLK 之间的关系如下：

波特率除数 = $\text{UARTCLK}/(16 \times \text{波特率}) = BRD_I + BRD_F$ 其中 BRD_I 为整数部分， BRD_F 为如图 60 所示由小数点分隔的小数部分。

图 60. 波特率除数。



可通过取所需波特率除数的小数部分并乘以 64（即，其中为 UARTFBRD 寄存器的位宽），再加 0.5 以进行四舍五入，计算该 6 位数 2^n ： $n = \text{integer}(BRD_F \times 64 + 0.5)$

内部生成一个时钟使能信号 Baud16，该信号为一串宽度为一个 UARTCLK 的脉冲，其平均频率为所需波特率的 16 倍。该信号随后被除以 16，作为发送时钟。波特率除数较低时，位周期较短；波特率除数较高时，位周期较长。

4.2.3.2.2 数据传输或接收

接收或发送的数据存储在两个32字节的FIFO中，但接收FIFO每个字符额外包含四位状态信息。对于发送，数据写入发送FIFO。若UART已启用，则会根据行控制寄存器UARTLCR_H中指定的参数启动数据帧的发送。数据持续传输，直至发送FIFO无数据。只要数据写入发送FIFO（即FIFO非空），BUSY信号即被置高，并在数据传输过程中保持高电平。仅当发送FIFO为空，且最后一个字符（包括停止位）已从移位寄存器发送完毕时，BUSY信号才会被置低。即使UART可能已不再启用，BUSY信号仍可能保持高电平。

对于每个数据样本，采集三次读数，并保留多数值。以下段落中定义了中间采样点，并在其两侧各取一个样本。

当接收器处于空闲状态（UARTRXD持续为1，处于标记状态）且在数据输入端检测到低电平（表示接收到起始位）时，由Baud16使能时钟的接收计数器启动计数，数据在UART模式下于计数器的第八个周期采样，或在SIR模式下于计数器的第四个周期采样，以适应较短的逻辑0脉冲（位周期中点）。

若Baud16的第八个周期时UARTRXD仍为低电平，则判定起始位有效，否则视为误起始位并予以忽略。

若起始位有效，依据编程设置的数据字符长度，随后每隔16个Baud16周期采样一次数据位（即一个位周期后采样）。如果启用了奇偶校验模式，则随后检查奇偶校验位。

最后，如果UARTRXD为高电平，则确认有效的停止位，否则发生帧错误。当接收到完整字时，数据存储于接收FIFO中，且与该字相关的任何错误位也一并存储。

4.2.3.2.3. 错误位

三个错误位存储于接收FIFO的第10至第8位，并关联至特定字符。另有一项溢出错误，存储于接收FIFO的第11位。

4.2.3.2.4. Overrun bit

The overrun bit is not associated with the character in the receive FIFO. The overrun error is set when the FIFO is full, and the next character is completely received in the shift register. The data in the shift register is overwritten, but it is not written into the FIFO. When an empty location is available in the receive FIFO, and another character is received, the state of the overrun bit is copied into the receive FIFO along with the received character. The overrun state is then cleared. [Table 422](#) lists the bit functions of the receive FIFO.

Table 422. Receive FIFO bit functions

FIFO bit	Function
11	Overrun indicator
10	Break error
9	Parity error
8	Framing error
7:0	Received data

4.2.3.2.5. Disabling the FIFOs

Additionally, you can disable the FIFOs. In this case, the transmit and receive sides of the UART have 1-byte holding registers (the bottom entry of the FIFOs). The overrun bit is set when a word has been received, and the previous one was not yet read. In this implementation, the FIFOs are not physically disabled, but the flags are manipulated to give the illusion of a 1-byte register. When the FIFOs are disabled, a write to the data register bypasses the holding register unless the transmit shift register is already in use.

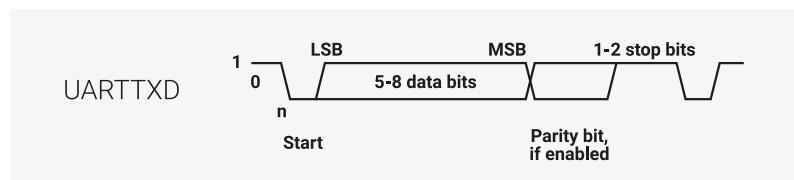
4.2.3.2.6. System and diagnostic loopback testing

You can perform loopback testing for UART data by setting the Loop Back Enable (LBE) bit to 1 in the Control Register, [UARTCR](#).

Data transmitted on UARTRXD is received on the UARTRXD input.

4.2.3.3. UART character frame

Figure 61. UART character frame.



4.2.4. UART hardware flow control

The hardware flow control feature is fully selectable, and enables you to control the serial data flow by using the nUARTRTS output and nUARTCTS input signals. [Figure 62](#) shows how two devices can communicate with each other using hardware flow control.

4.2.3.2.4. 溢出位

溢出位不与接收FIFO中的字符关联。当FIFO已满且移位寄存器中完整接收下一字符时，溢出错误位被置位。移位寄存器中的数据将被覆盖，但不会写入FIFO。当接收FIFO有空位且接收到另一字符时，溢出位状态会随所接收字符一并复制入接收FIFO。随后清除溢出状态。表422列出了接收FIFO的位功能。

表422。接收
FIFO位功能

FIFO位	功能
11	溢出指示符
10	断帧错误
9	奇偶校验错误
8	帧错误
7:0	接收数据

4.2.3.2.5. 禁用FIFO

此外，您可以禁用FIFO。在此情况下，UART的发送和接收端各配备1字节保持寄存器（即FIFO的最低条目）。当接收到一个字且前一个字尚未读取时，溢出位将被置位。在本实现中，FIFO未被物理禁用，而是通过操作标志位以模拟1字节寄存器的行为。FIFO禁用时，对数据寄存器的写操作会绕过保持寄存器，除非发送移位寄存器已在使用。

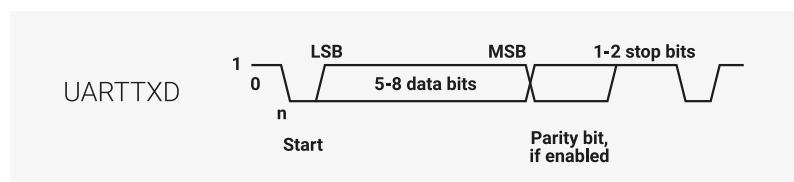
4.2.3.2.6. 系统及诊断环回测试

您可以通过将控制寄存器 UARTCR 中的循环回环使能（LBE）位设置为 1 来执行 UART 数据的回环测试，[UARTCR](#)。

在 UARTRXD 上发送的数据将在 UARTRXD 输入端接收。

4.2.3.3 UART 字符帧

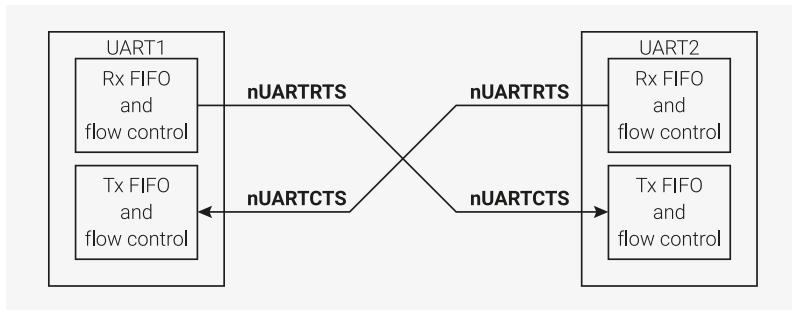
图 61 UART
字符帧。



4.2.4. UART 硬件流控制

硬件流控功能完全可选，允许您通过 nUARTRTS 输出和 nUARTCTS 输入信号控制串行数据流。图 62 展示了两个设备如何利用硬件流控进行通信。

Figure 62. Hardware flow control between two similar devices.



When the RTS flow control is enabled, nUARTRTS is asserted until the receive FIFO is filled up to the programmed watermark level. When the CTS flow control is enabled, the transmitter can only transmit data when nUARTCTS is asserted.

The hardware flow control is selectable using the RTSEn and CTSEn bits in the Control Register, [UARTCR](#). Table 423 lists how you must set the bits to enable RTS and CTS flow control both simultaneously, and independently.

Table 423. Control bits to enable and disable hardware flow control.

UARTCR Register bits		
CTSEn	RTSEn	Description
1	1	Both RTS and CTS flow control enabled
1	0	Only CTS flow control enabled
0	1	Only RTS flow control enabled
0	0	Both RTS and CTS flow control disabled

NOTE

When RTS flow control is enabled, the software cannot use the RTSEn bit in the Control Register, [UARTCR](#), to control the status of nUARTRTS.

4.2.4.1. RTS flow control

The RTS flow control logic is linked to the programmable receive FIFO watermark levels. When RTS flow control is enabled, the nUARTRTS is asserted until the receive FIFO is filled up to the watermark level. When the receive FIFO watermark level is reached, the nUARTRTS signal is deasserted, indicating that there is no more room to receive any more data. The transmission of data is expected to cease after the current character has been transmitted.

The nUARTRTS signal is reasserted when data has been read out of the receive FIFO so that it is filled to less than the watermark level. If RTS flow control is disabled and the UART is still enabled, then data is received until the receive FIFO is full, or no more data is transmitted to it.

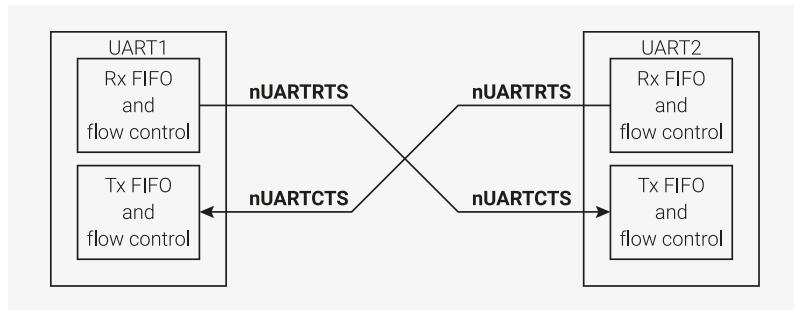
4.2.4.2. CTS flow control

If CTS flow control is enabled, then the transmitter checks the nUARTCTS signal before transmitting the next byte. If the nUARTCTS signal is asserted, it transmits the byte otherwise transmission does not occur.

The data continues to be transmitted while nUARTCTS is asserted, and the transmit FIFO is not empty. If the transmit FIFO is empty and the nUARTCTS signal is asserted no data is transmitted.

If the nUARTCTS signal is deasserted and CTS flow control is enabled, then the current character transmission is completed before stopping. If CTS flow control is disabled and the UART is enabled, then the data continues to be transmitted until the transmit FIFO is empty.

图 62。两个相似设备之间的硬件流控制。



启用 RTS 流控后，nUARTRTS 信号将保持有效，直到接收 FIFO 达到预设水位线。启用 CTS 流控后，只有在 nUARTCTS 信号有效时，发送端才能传输数据。

硬件流控可通过控制寄存器 UARTCR 中的 RTSEn 和 CTSEn 位选择。表 423 列出了同时启用或分别启用 RTS 和 CTS 流控时各个位的设置方法。

表 423。用于启用和禁用硬件流控制的控制位。

UARTCR 寄存器位		
CTSEn	RTSEn	描述
1	1	同时启用 RTS 和 CTS 流控制
1	0	仅启用 CTS 流控制
0	1	仅启用 RTS 流控制
0	0	同时禁用 RTS 和 CTS 流控制

① 注意

启用 RTS 流控制时，软件无法使用控制寄存器 UARTCR 中的 RTSEn 位来控制 nUARTRTS 的状态。

4.2.4.1 RTS 流控制

RTS 流控制逻辑与可编程接收 FIFO 的水位线级别相关联。启用 RTS 流控制时，nUARTRTS 信号被激活，直到接收 FIFO 填满至水位线水平。当接收 FIFO 达到水位线水平时，nUARTRTS 信号被取消激活，表示接收缓冲区已无更多数据接收空间。预计当前字符传输完成后，数据传输将停止。

当接收 FIFO 中的数据被读取，使其填充量降至低于水位线时，nUARTRTS 信号将重新断言。如果 RTS 流控被禁用且 UART 仍处于使能状态，则数据将继续接收，直到接收 FIFO 已满或没有更多数据传入。

4.2.4.2. CTS流控

如果启用 CTS 流控，则发送器在发送下一个字节前会检测 nUARTCTS 信号。若 nUARTCTS 信号被断言，则发送该字节；否则不进行传输。

只要 nUARTCTS 被断言且发送 FIFO 非空，数据将持续传输。如果发送 FIFO 为空且 nUARTCTS 信号被断言，则不发送数据。

如果 nUARTCTS 信号被取消断言且启用 CTS 流控，则当前字符传输完成后停止传输。如果 CTS 流控被禁用且 UART 被启用，则数据继续传输直至发送 FIFO 为空。

4.2.5. UART DMA Interface

The UART provides an interface to connect to a DMA controller. The DMA operation of the UART is controlled using the DMA Control Register, [UARTDMACR](#). The DMA interface includes the following signals:

For receive:

UARTRXDMASREQ

Single character DMA transfer request, asserted by the UART. For receive, one character consists of up to 12 bits. This signal is asserted when the receive FIFO contains at least one character.

UARTRXDMABREQ

Burst DMA transfer request, asserted by the UART. This signal is asserted when the receive FIFO contains more characters than the programmed watermark level. You can program the watermark level for each FIFO using the Interrupt FIFO Level Select Register, [UARTIFLS](#)

UARTRXDMACLR

DMA request clear, asserted by a DMA controller to clear the receive request signals. If DMA burst transfer is requested, the clear signal is asserted during the transfer of the last data in the burst.

For transmit:

UARTTXDMASREQ

Single character DMA transfer request, asserted by the UART. For transmit one character consists of up to eight bits. This signal is asserted when there is at least one empty location in the transmit FIFO.

UARTTXDMABREQ

Burst DMA transfer request, asserted by the UART. This signal is asserted when the transmit FIFO contains less characters than the watermark level. You can program the watermark level for each FIFO using the Interrupt FIFO Level Select Register, [UARTIFLS](#).

UARTTXDMACLR

DMA request clear, asserted by a DMA controller to clear the transmit request signals. If DMA burst transfer is requested, the clear signal is asserted during the transfer of the last data in the burst.

The burst transfer and single transfer request signals are not mutually exclusive, they can both be asserted at the same time. For example, when there is more data than the watermark level in the receive FIFO, the burst transfer request and the single transfer request are asserted. When the amount of data left in the receive FIFO is less than the watermark level, the single request only is asserted. This is useful for situations where the number of characters left to be received in the stream is less than a burst.

For example, if 19 characters have to be received and the watermark level is programmed to be four. The DMA controller then transfers four bursts of four characters and three single transfers to complete the stream.

NOTE

For the remaining three characters the UART cannot assert the burst request.

Each request signal remains asserted until the relevant DMACLR signal is asserted. After the request clear signal is deasserted, a request signal can become active again, depending on the conditions described previously. All request signals are deasserted if the UART is disabled or the relevant DMA enable bit, TXDMAE or RXDMAE, in the DMA Control Register, [UARTDMACR](#), is cleared.

If you disable the FIFOs in the UART then it operates in character mode and only the DMA single transfer mode can operate, because only one character can be transferred to, or from the FIFOs at any time. UARTRXDMASREQ and UARTTXDMASREQ are the only request signals that can be asserted. See the Line Control Register, [UARTLCR_H](#), for information about disabling the FIFOs.

When the UART is in the FIFO enabled mode, data transfers can be made by either single or burst transfers depending on the programmed watermark level and the amount of data in the FIFO. [Table 424](#) lists the trigger points for UARTRXDMABREQ and UARTTXDMABREQ depending on the watermark level, for the transmit and receive FIFOs.

4.2.5. UART DMA 接口

UART 提供一个接口，用于连接 DMA 控制器。UART 的 DMA 操作通过 DMA 控制寄存器 UARTDMACR 进行控制。DMA 接口包含以下信号：

接收：

UARTRXDMASREQ

单字符 DMA 传输请求，由 UART 断言。对于接收，单字符最多包含 12 位。
当接收 FIFO 至少含有一个字符时，该信号被断言。

UARTRXDMABREQ

突发 DMA 传输请求，由 UART 断言。当接收 FIFO 中的字符数超过编程设定的水位线时，该信号被断言。您可以通过中断 FIFO 水平选择寄存器 UARTIFLS 为每个 FIFO 编程水位线。

UARTRXDMACLR

DMA 请求清除信号，由 DMA 控制器断言，用以清除接收请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

发送：

UARTTXDMASREQ

单字符 DMA 传输请求，由 UART 发出。传输时，一个字符由最多八位组成。当发送 FIFO 中至少存在一个空位时，该信号被置位。

UARTTXDMABREQ

突发 DMA 传输请求，由 UART 发出。当发送 FIFO 中的字符数少于水位线时，该信号被置位。您可以通过中断 FIFO 级别选择寄存器 UARTIFLS 为每个 FIFO 设置水位线级别。

UARTTXDMACLR

DMA 请求清除信号，由 DMA 控制器发出，用以清除发送请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

突发传输请求信号与单次传输请求信号并非互斥，可同时置位。例如，当接收 FIFO 中的数据量多于水位线时，突发传输请求与单次传输请求均被置位。当接收 FIFO 中剩余数据量少于水位线时，仅单次传输请求被置位。此功能适用于串流中剩余待接收字符数少于突发传输量的情况。

例如，若需接收 19 个字符且水印级别设置为四。随后，DMA 控制器执行四次四字符突发传输及三次单字符传输以完成数据流。

i 注意

对于剩余的三个字符，UART 无法发出突发请求信号。

每个请求信号保持有效状态，直至相应的 DMA CLR 信号被置位。在请求清除信号取消置位后，请求信号可根据前述条件重新激活。若 UART 被禁用，或 DMA 控制寄存器 UARTDMACR 中的相关 DMA 使能位 TXDMAE 或 RXDMAE 被清除，所有请求信号均被取消置位。

若禁用 UART 中的 FIFO，则其工作于字符模式，仅支持 DMA 单字符传输模式，因每次仅能传输一个字符至或从 FIFO。UARTRXDMASREQ 与 UARTTXDMASREQ 为唯一可置位的请求信号。有关禁用 FIFO 的信息，请参见线路控制寄存器 UARTLCR_LH。

当 UART 处于启用 FIFO 模式时，数据传输可根据编程的水位线水平及 FIFO 中的数据量，通过单次或突发传输完成。表 4-24 列出了根据水位线水平，针对发送和接收 FIFO 的 UARTRXDMABREQ 和 UARTTXDMABREQ 触发点。

Table 424. DMA trigger points for the transmit and receive FIFOs.

Watermark level	Burst length	
	Transmit (number of empty locations)	Receive (number of filled locations)
1/8	28	4
1/4	24	8
1/2	16	16
3/4	8	24
7/8	4	28

In addition, the DMAONERR bit in the DMA Control Register, [UARTDMACR](#), supports the use of the receive error interrupt, UARTEINTR. It enables the DMA receive request outputs, UARTRXDMASREQ or UARTRXDMABREQ, to be masked out when the UART error interrupt, UARTEINTR, is asserted. The DMA receive request outputs remain inactive until the UARTEINTR is cleared. The DMA transmit request outputs are unaffected.

Figure 63. DMA transfer waveforms.

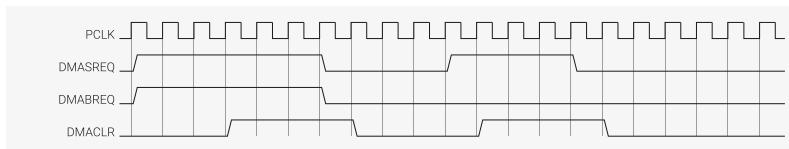


Figure 63 shows the timing diagram for both a single transfer request and a burst transfer request with the appropriate DMACLR signal. The signals are all synchronous to PCLK. For the sake of clarity it is assumed that there is no synchronization of the request signals in the DMA controller.

4.2.6. Interrupts

There are eleven maskable interrupts generated in the UART. On RP2040, only the combined interrupt output, [UARTINTR](#), is connected.

You can enable or disable the individual interrupts by changing the mask bits in the Interrupt Mask Set/Clear Register, [UARTIMSC](#). Setting the appropriate mask bit HIGH enables the interrupt.

Provision of individual outputs and the combined interrupt output, enables you to use either a global interrupt service routine, or modular device drivers to handle interrupts.

The transmit and receive dataflow interrupts UARTRXINTR and UARTRXINTR have been separated from the status interrupts. This enables you to use UARTRXINTR and UARTRXINTR so that data can be read or written in response to the FIFO trigger levels.

The error interrupt, UARTEINTR, can be triggered when there is an error in the reception of data. A number of error conditions are possible.

The modem status interrupt, [UARTMSINTR](#), is a combined interrupt of all the individual modem status signals.

The status of the individual interrupt sources can be read either from the Raw Interrupt Status Register, [UARTRIS](#), or from the Masked Interrupt Status Register, [UARTMIS](#).

4.2.6.1. [UARTMSINTR](#)

The modem status interrupt is asserted if any of the modem status signals (nUARTCTS, nUARTDCD, nUARTDSR, and nUARTRI) change. It is cleared by writing a 1 to the corresponding bit(s) in the Interrupt Clear Register, [UARTICR](#), depending on the modem status signals that generated the interrupt.

表 424。发送和接收 FIFO 的 DMA 触发点。

水位线	突发长度	
	发送（空闲位置数）	接收（填充位置数量）
1/8	28	4
1/4	24	8
1/2	16	16
3/4	8	24
7/8	4	28

此外，DMA 控制寄存器 UARTDMACR 中的 DMAONERR 位支持接收错误中断 UARTEINTR 的使用。当 UART 错误中断 UARTEINTR 被触发时，该位可屏蔽 DMA 接收请求输出 UARTRXDMASREQ 或 UARTRXDMABREQ。DMA 接收请求输出保持非激活状态，直至清除 UARTEINTR。DMA 发送请求输出不受影响。

图63. DMA 传输波形。

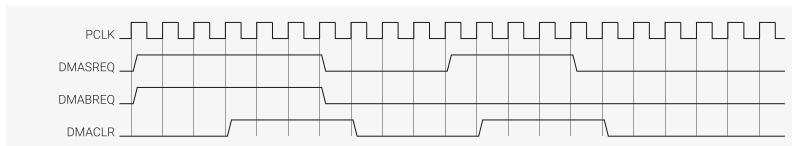


图63显示了单次传输请求及携带相应DMACLR信号的突发传输请求的时序图。所有信号均与PCLK同步。为简明起见，假设DMA控制器内的请求信号未进行同步处理。

4.2.6. 中断

UART产生十一种可屏蔽中断。在RP2040上，仅连接了组合中断输出UARTINTR。

您可通过修改中断屏蔽设置/清除寄存器UARTIMSC中的屏蔽位来启用或禁用各个中断。相应屏蔽位置高即启用该中断。

单独中断输出及组合中断输出的提供，使您能够使用全局中断服务程序或模块化设备驱动程序来处理中断。

发送和接收数据流中断UARTRXINTR与UARTTXINTR已与状态中断分离。此功能使您能够使用 UARTRXINTR 和 UARTTXINTR，以便根据 FIFO 触发电平读取或写入数据。

当接收数据发生错误时，错误中断 UARTEINTR 会被触发。可能存在多种错误情况。

调制解调器状态中断 UARTMSINTR 是由所有独立调制解调器状态信号组合而成的中断。

各个中断源的状态可以从原始中断状态寄存器 UARTRIS 中读取，
也可以从屏蔽中断状态寄存器 UARTMIS 中读取。

4.2.6.1. UARTMSINTR

当任何调制解调器状态信号（nUARTCTS、nUARTDCD、nUARTDSR 和 nUARTRI）发生变化时，调制解调器状态中断即被置位。根据引发中断的调制解调器状态信号，通过向中断清除寄存器 UARTRICR 中相应的位写入 1 来清除该中断。

4.2.6.2. UARTRXINTR

The receive interrupt changes state when one of the following events occurs:

- If the FIFOs are enabled and the receive FIFO reaches the programmed trigger level. When this happens, the receive interrupt is asserted HIGH. The receive interrupt is cleared by reading data from the receive FIFO until it becomes less than the trigger level, or by clearing the interrupt.
- If the FIFOs are disabled (have a depth of one location) and data is received thereby filling the location, the receive interrupt is asserted HIGH. The receive interrupt is cleared by performing a single read of the receive FIFO, or by clearing the interrupt.

4.2.6.3. UARTTXINTR

The transmit interrupt changes state when one of the following events occurs:

- If the FIFOs are enabled and the transmit FIFO is equal to or lower than the programmed trigger level then the transmit interrupt is asserted HIGH. The transmit interrupt is cleared by writing data to the transmit FIFO until it becomes greater than the trigger level, or by clearing the interrupt.
- If the FIFOs are disabled (have a depth of one location) and there is no data present in the transmitters single location, the transmit interrupt is asserted HIGH. It is cleared by performing a single write to the transmit FIFO, or by clearing the interrupt.

To update the transmit FIFO you must:

- Write data to the transmit FIFO, either prior to enabling the UART and the interrupts, or after enabling the UART and interrupts.

i NOTE

The transmit interrupt is based on a transition through a level, rather than on the level itself. When the interrupt and the UART is enabled before any data is written to the transmit FIFO the interrupt is not set. The interrupt is only set, after written data leaves the single location of the transmit FIFO and it becomes empty.

4.2.6.4. UARTRTINTR

The receive timeout interrupt is asserted when the receive FIFO is not empty, and no more data is received during a 32-bit period. The receive timeout interrupt is cleared either when the FIFO becomes empty through reading all the data (or by reading the holding register), or when a 1 is written to the corresponding bit of the Interrupt Clear Register, [UARTICR](#).

4.2.6.5. UARTEINTR

The error interrupt is asserted when an error occurs in the reception of data by the UART. The interrupt can be caused by a number of different error conditions:

- framing
- parity
- break
- overrun.

You can determine the cause of the interrupt by reading the Raw Interrupt Status Register, [UARTRIS](#), or the Masked Interrupt Status Register, [UARTMIS](#). It can be cleared by writing to the relevant bits of the Interrupt Clear Register, [UARTICR](#) (bits 7 to 10 are the error clear bits).

4.2.6.2. UARTRXINTR

当发生以下任一事件时，接收中断状态将改变：

- 如果启用了FIFO且接收FIFO达到预设的触发级别。发生此情况时，接收中断被置为高电平。接收中断通过从接收FIFO读取数据直到其低于触发级别，或通过清除中断予以清除。
- 如果禁用FIFO（深度为一个位置），且接收到数据使该位置被填满，则接收中断被置为高电平。接收中断通过对接收FIFO执行单次读取，或通过清除中断予以清除。

4.2.6.3. UARTTXINTR

当发生以下任一事件时，发送中断状态将改变：

- 如果启用了FIFO且发送FIFO低于或等于预设的触发级别，则发送中断被置为高电平。发送中断通过向发送FIFO写入数据直到其超过触发级别，或通过清除中断予以清除。
- 如果FIFO被禁用（深度为一个位置）且发射器的单个位置中无数据，则发送中断被置为高电平。该中断可通过对发送FIFO执行单次写操作或清除中断来清除。

要更新发送FIFO，必须：

- 向发送FIFO写入数据，可以在启用UART和中断之前，或启用UART和中断之后执行。

i 注意

发送中断基于电平的跳变触发，而非电平本身。当中断与UART在向发送FIFO写入任何数据之前被启用时，中断不会被置位。只有在写入数据离开发送FIFO的单个位置且该位置变为空时，中断才会被置位。

4.2.6.4. UARTRTINTR

当接收FIFO非空且在32位周期内未接收到更多数据时，接收超时中断被触发。接收超时中断将在FIFO通过读取所有数据（或读取保持寄存器）变为空，或写1至中断清除寄存器UARTICR的对应位时被清除。

4.2.6.5. UARTEINTR

当UART接收数据发生错误时，会触发错误中断。该中断可能由多种不同的错误情况引起：

- 帧错误
- 奇偶校验错误
- 断帧
- 溢出错误。

您可以通过读取原始中断状态寄存器（Raw Interrupt Status Register，UARTRIS）或屏蔽中断状态寄存器（Masked Interrupt Status Register，UARTMIS）来确定中断原因。中断可通过向中断清除寄存器（Interrupt Clear Register，UARTICR）相关位写入数据来清除（第7至10位为错误清除位）。

4.2.6.6. UARTINTR

The interrupts are also combined into a single output, that is an OR function of the individual masked sources. You can connect this output to a system interrupt controller to provide another level of masking on a individual peripheral basis.

The combined UART interrupt is asserted if any of the individual interrupts are asserted and enabled.

4.2.7. Programmer's Model

The SDK provides a `uart_init` function to configure the UART with a particular baud rate. Once the UART is initialised, the user must configure a GPIO pin as `UART_TX` and `UART_RX`. See [Section 2.19.5.1](#) for more information on selecting a GPIO function.

To initialise the UART, the `uart_init` function takes the following steps:

- Deassert the reset
- Enable `clk_peri`
- Set enable bits in the control register
- Enable the FIFOs
- Set the baud rate divisors
- Set the format

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c Lines 42 - 92

```

42 uint uart_init(uart_inst_t *uart, uint baudrate) {
43     invalid_params_if(HARDWARE_UART, uart != uart0 && uart != uart1);
44
45     if (uart_clock_get_hz(uart) == 0) {
46         return 0;
47     }
48
49     uart_reset(uart);
50     uart_unreset(uart);
51
52     uart_set_translate_crlf(uart, PICO_UART_DEFAULT_CRLF);
53
54     // Any LCR writes need to take place before enabling the UART
55     uint baud = uart_set_baudrate(uart, baudrate);
56
57     // inline the uart_set_format() call, as we don't need the CR disable/re-enable
58     // protection, and also many people will never call it again, so having
59     // the generic function is not useful, and much bigger than this inlined
60     // code which is only a handful of instructions.
61     //
62     // The UART_UARTLCR_H_FEN_BITS setting is combined as well as it is the same register
63 #ifdef 0
64     uart_set_format(uart, 8, 1, UART_PARITY_NONE);
65     // Enable FIFOs (must be before setting UARLEN, as this is an LCR access)
66     hw_set_bits(&uart_get_hw(uart)->lcr_h, UART_UARTLCR_H_FEN_BITS);
67 #else
68     uint data_bits = 8;
69     uint stop_bits = 1;
70     uint parity = UART_PARITY_NONE;
71     hw_write_masked(&uart_get_hw(uart)->lcr_h,
72                     ((data_bits - 5u) << UART_UARTLCR_H_WLEN_LSB) |
73                     ((stop_bits - 1u) << UART_UARTLCR_H_STP2_LSB) |
74                     (bool_to_bit(parity != UART_PARITY_NONE) << UART_UARTLCR_H_PEN_LSB) |
75                     (bool_to_bit(parity == UART_PARITY_EVEN) << UART_UARTLCR_H_EPS_LSB) |

```

4.2.6.6. UARTINTR

这些中断也会被合并为一个输出，该输出为各个屏蔽中断源的逻辑或。您可以将此输出连接至系统中断控制器，以针对单个外设实现另一级别的屏蔽。

只要任一单独中断被触发且启用，合并的UART中断即被触发。

4.2.7. 程序员模型

SDK提供了一个`uart_init`函数，用于配置特定波特率的UART。UART初始化完成后，用户必须将GPIO引脚配置为`UART_T`与`UART_RX`。有关GPIO功能选择的更多信息，请参见第2.19.5.1节。

初始化UART时，`uart_init`函数按以下步骤执行：

- 取消复位信号
- 启用`clk_peri`
- 在控制寄存器中设置使能位
- 启用FIFO
- 设置波特率分频器
- 配置格式

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c 第42至92行

```

42 uint uart_init(uart_inst_t *uart, uint baudrate) {
43     invalid_params_if(HARDWARE_UART, uart != uart0 && uart != uart1);
44
45     if (uart_clock_get_hz(uart) == 0) {
46         return 0;
47     }
48
49     uart_reset(uart);
50     uart_unreset(uart);
51
52     uart_set_translate_crlf(uart, PICO_UART_DEFAULT_CRLF);
53
54     // 所有LCR寄存器写入操作必须在启用UART之前完成
55     uint baud = uart_set_baudrate(uart, baudrate);
56
57     // 内联uart_set_format()调用，因为我们不需要对CR进行禁用/重新启用
58     // 保护，且许多人可能永远不会再调用它，
59     // 因此通用函数没有实用价值，且其内联代码远小于函数本身，
60     // 仅包含少量指令。
61
62     // UART_UARTLCR_H_FEN_BITS设置合并处理，因为它们位于同一寄存器
63 #ifdef 0
64     uart_set_format(uart, 8, 1, UART_PARITY_NONE);
65     // 启用FIFO（必须在设置UARTEN之前完成，因为这是一次LCR寄存器访问）
66     hw_set_bits(&uart_get_hw(uart)->lcr_h, UART_UARTLCR_H_FEN_BITS);
67 #else
68     uint data_bits = 8;
69     uint stop_bits = 1;
70     uint parity = UART_PARITY_NONE;
71     hw_write_masked(&uart_get_hw(uart)->lcr_h,
72                     ((data_bits - 5u) << UART_UARTLCR_H_WLEN_LSB) |
73                     ((stop_bits - 1u) << UART_UARTLCR_H_STP2_LSB) |
74                     (bool_to_bit(parity != UART_PARITY_NONE) << UART_UARTLCR_H_PEN_LSB) |
75                     (bool_to_bit(parity == UART_PARITY_EVEN) << UART_UARTLCR_H_EPS_LSB) |

```

```

76         UART_UARTLCR_H_FEN_BITS,
77         UART_UARTLCR_H_WLEN_BITS | UART_UARTLCR_H_STP2_BITS |
78         UART_UARTLCR_H_PEN_BITS | UART_UARTLCR_H_EPS_BITS |
79         UART_UARTLCR_H_FEN_BITS);
80 #endif
81
82 // Enable the UART, both TX and RX
83 uart_get_hw(uart)->cr = UART_UARTCR_UARTEN_BITS | UART_UARTCR_TXE_BITS |
84     UART_UARTCR_RXE_BITS;
85 // Always enable DREQ signals -- no harm in this if DMA is not listening
86 uart_get_hw(uart)->dmacr = UART_UARTDMACR_TXDMAE_BITS | UART_UARTDMACR_RXDMAE_BITS;
87
88 return baud;
89 }
```

4.2.7.1. Baud Rate Calculation

The uart baud rate is derived from dividing `clk_peri`.

If the required baud rate is 115200 and `UARTCLK` = 125MHz then:

$$\text{Baud Rate Divisor} = (125 * 10^6) / (16 * 115200) \approx 67.817$$

Therefore, `BRDI` = 67 and `BRDF` = 0.817,

Therefore, fractional part, $m = \text{integer}((0.817 * 64) + 0.5) = 52$

$$\text{Generated baud rate divider} = 67 + 52/64 = 67.8125$$

$$\text{Generated baud rate} = (125 * 10^6) / (16 * 67.8125) \approx 115207$$

$$\text{Error} = (\text{abs}(115200 - 115207) / 115200) * 100 \approx 0.006\%$$

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c Lines 155 - 180

```

155 uint uart_set_baudrate(uart_inst_t *uart, uint baudrate) {
156     invalid_params_if(HARDWARE_UART, baudrate == 0);
157     uint32_t baud_rate_div = (8 * uart_clock_get_hz(uart) / baudrate) + 1;
158     uint32_t baud_ibrd = baud_rate_div >> 7;
159     uint32_t baud_fbrd;
160
161     if (baud_ibrd == 0) {
162         baud_ibrd = 1;
163         baud_fbrd = 0;
164     } else if (baud_ibrd >= 65535) {
165         baud_ibrd = 65535;
166         baud_fbrd = 0;
167     } else {
168         baud_fbrd = (baud_rate_div & 0x7f) >> 1;
169     }
170
171     uart_get_hw(uart)->ibrd = baud_ibrd;
172     uart_get_hw(uart)->fbrd = baud_fbrd;
173
174 // PL011 needs a (dummy) LCR_H write to latch in the divisors.
175 // We don't want to actually change LCR_H contents here.
176     uart_write_lcr_bits_masked(uart, 0, 0);
177
178 // See datasheet
179     return (4 * uart_clock_get_hz(uart)) / (64 * baud_ibrd + baud_fbrd);
180 }
```

```

76         UART_UARTLCR_H_FEN_BITS,
77         UART_UARTLCR_H_WLEN_BITS | UART_UARTLCR_H_STP2_BITS |
78         UART_UARTLCR_H_PEN_BITS | UART_UARTLCR_H_EPS_BITS |
79         UART_UARTLCR_H_FEN_BITS);
80 #endif
81
82 // 启用 UART，包括发送（TX）和接收（RX）
83 uart_get_hw(uart)->cr = UART_UARTCR_UARTEN_BITS | UART_UARTCR_TXE_BITS |
84     UART_UARTCR_RXE_BITS;
85 // 始终启用 DREQ 信号——若 DMA 未监听，启用无害
86 uart_get_hw(uart)->dmacr = UART_UARTDMACR_TXDMAE_BITS | UART_UARTDMACR_RXDMAE_BITS;
87
88 return baud;
89 }
```

4.2.7.1. 波特率计算

UART 波特率是通过除以 `clk_peri` 计算得出。

若所需波特率为 115200 且 UARTCLK 为 125MHz，则：

$$\text{波特率除数} = (125 * 10^6) / (16 * 115200) \approx 67.817$$

因此，BRDI = 67，BRDF = 0.817，

因此，小数部分 m = integer((0.817 * 64) + 0.5) = 52

生成的波特率分频值 = $67 + 52/64 = 67.8125$

$$\text{生成的波特率} = (125 * 10^6) / (16 * 67.8125) \approx 115207$$

$$\text{误差} = (\text{abs}(115200 - 115207) / 115200) * 100 \approx 0.006\%$$

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_uart/uart.c，第155至180行

```

155 uint uart_set_baudrate(uart_inst_t *uart, uint baudrate) {
156     invalid_params_if(HARDWARE_UART, baudrate == 0);
157     uint32_t baud_rate_div = (8 * uart_clock_get_hz(uart) / baudrate) + 1;
158     uint32_t baud_ibrd = baud_rate_div >> 7;
159     uint32_t baud_fbrd;
160
161     if (baud_ibrd == 0) {
162         baud_ibrd = 1;
163         baud_fbrd = 0;
164     } else if (baud_ibrd >= 65535) {
165         baud_ibrd = 65535;
166         baud_fbrd = 0;
167     } else {
168         baud_fbrd = (baud_rate_div & 0x7f) >> 1;
169     }
170
171     uart_get_hw(uart)->ibrd = baud_ibrd;
172     uart_get_hw(uart)->fbrd = baud_fbrd;
173
174 // PL011 需要一个（伪）LCR_H 写入以锁存分频值。
175 // 此处不应实际更改 LCR_H 的内容。
176     uart_write_lcr_bits_masked(uart, 0, 0);
177
178 // 参见数据手册
179     return (4 * uart_clock_get_hz(uart)) / (64 * baud_ibrd + baud_fbrd);
180 }
```

4.2.8. List of Registers

The UART0 and UART1 registers start at base addresses of `0x40034000` and `0x40038000` respectively (defined as `UART0_BASE` and `UART1_BASE` in SDK).

Table 425. List of
UART registers

Offset	Name	Info
0x000	UARTDR	Data Register, UARTDR
0x004	UARTRSR	Receive Status Register/Error Clear Register, UARTRSR/UARTECR
0x018	UARTFR	Flag Register, UARTFR
0x020	UARTILPR	IrDA Low-Power Counter Register, UARTILPR
0x024	UARTIBRD	Integer Baud Rate Register, UARTIBRD
0x028	UARTFBRD	Fractional Baud Rate Register, UARTFBRD
0x02c	UARTLCR_H	Line Control Register, UARTLCR_H
0x030	UARTCR	Control Register, UARTCR
0x034	UARTIFLS	Interrupt FIFO Level Select Register, UARTIFLS
0x038	UARTIMSC	Interrupt Mask Set/Clear Register, UARTIMSC
0x03c	UARTRIS	Raw Interrupt Status Register, UARTRIS
0x040	UARTMIS	Masked Interrupt Status Register, UARTMIS
0x044	UARTICR	Interrupt Clear Register, UARTICR
0x048	UARTDMACR	DMA Control Register, UARTDMACR
0xfe0	UARTPERIPHID0	UARTPeriphID0 Register
0xfe4	UARTPERIPHID1	UARTPeriphID1 Register
0xfe8	UARTPERIPHID2	UARTPeriphID2 Register
0xfc	UARTPERIPHID3	UARTPeriphID3 Register
0xff0	UARTPCELLID0	UARTPCellID0 Register
0xff4	UARTPCELLID1	UARTPCellID1 Register
0xff8	UARTPCELLID2	UARTPCellID2 Register
0ffc	UARTPCELLID3	UARTPCellID3 Register

UART: UARTDR Register

Offset: 0x000

Description

Data Register, UARTDR

Table 426. UARTDR
Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	OE: Overrun error. This bit is set to 1 if data is received and the receive FIFO is already full. This is cleared to 0 once there is an empty space in the FIFO and a new character can be written to it.	RO	-

4.2.8. 寄存器列表

UART0 和 UART1 寄存器的基地址分别为 `0x40034000` 和 `0x40038000` (在 SDK 中定义为 `UART0_BASE` 和 `UART1_BASE`)

◦

表 425. UART 寄存器列表

偏移量	名称	说明
0x000	<code>UARTDR</code>	数据寄存器, <code>UARTDR</code>
0x004	<code>UARTRSR</code>	接收状态寄存器／错误清除寄存器 <code>UARTRSR/UARTECR</code>
0x018	<code>UARTFR</code>	标志寄存器, <code>UARTFR</code>
0x020	<code>UARTILPR</code>	红外低功耗计数器寄存器, <code>UARTILPR</code>
0x024	<code>UARTIBRD</code>	整数波特率寄存器, <code>UARTIBRD</code>
0x028	<code>UARTFBRD</code>	分数波特率寄存器, <code>UARTFBRD</code>
0x02c	<code>UARTLCR_H</code>	线路控制寄存器, <code>UARTLCR_H</code>
0x030	<code>UARTCR</code>	控制寄存器, <code>UARTCR</code>
0x034	<code>UARTIFLS</code>	中断 FIFO 水平选择寄存器, <code>UARTIFLS</code>
0x038	<code>UARTIMSC</code>	中断屏蔽设置/清除寄存器, <code>UARTIMSC</code>
0x03c	<code>UARTRIS</code>	原始中断状态寄存器, <code>UARTRIS</code>
0x040	<code>UARTMIS</code>	掩码中断状态寄存器, <code>UARTMIS</code>
0x044	<code>UARTICR</code>	中断清除寄存器, <code>UARTICR</code>
0x048	<code>UARTDMACR</code>	DMA 控制寄存器, <code>UARTDMACR</code>
0xfe0	<code>UARTPERIPHID0</code>	UARTPeriphID0 寄存器
0xfe4	<code>UARTPERIPHID1</code>	UARTPeriphID1 寄存器
0xfe8	<code>UARTPERIPHID2</code>	UARTPeriphID2 寄存器
0xfc	<code>UARTPERIPHID3</code>	UARTPeriphID3 寄存器
0xff0	<code>UARTPCELLID0</code>	UARTPCellID0 寄存器
0xff4	<code>UARTPCELLID1</code>	UARTPCellID1 寄存器
0xff8	<code>UARTPCELLID2</code>	UARTPCellID2 寄存器
0ffc	<code>UARTPCELLID3</code>	UARTPCellID3 寄存器

UART：UARTDR 寄存器

偏移: 0x000

描述

数据寄存器, `UARTDR`

表 426. `UARTDR` 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	OE: 溢出错误。当接收到数据且接收 FIFO 已满时, 该位置为 1。一旦 FIFO 有空位且可写入新字符, 该位清零为 0。	只读	-

Bits	Description	Type	Reset
10	BE: Break error. This bit is set to 1 if a break condition was detected, indicating that the received data input was held LOW for longer than a full-word transmission time (defined as start, data, parity and stop bits). In FIFO mode, this error is associated with the character at the top of the FIFO. When a break occurs, only one 0 character is loaded into the FIFO. The next character is only enabled after the receive data input goes to a 1 (marking state), and the next valid start bit is received.	RO	-
9	PE: Parity error. When set to 1, it indicates that the parity of the received data character does not match the parity that the EPS and SPS bits in the Line Control Register, UARTLCR_H. In FIFO mode, this error is associated with the character at the top of the FIFO.	RO	-
8	FE: Framing error. When set to 1, it indicates that the received character did not have a valid stop bit (a valid stop bit is 1). In FIFO mode, this error is associated with the character at the top of the FIFO.	RO	-
7:0	DATA: Receive (read) data character. Transmit (write) data character.	RWF	-

UART: UARTRSR Register

Offset: 0x004

Description

Receive Status Register/Error Clear Register, UARTRSR/UARTECR

Table 427. UARTRSR Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	OE: Overrun error. This bit is set to 1 if data is received and the FIFO is already full. This bit is cleared to 0 by a write to UARTECR. The FIFO contents remain valid because no more data is written when the FIFO is full, only the contents of the shift register are overwritten. The CPU must now read the data, to empty the FIFO.	WC	0x0
2	BE: Break error. This bit is set to 1 if a break condition was detected, indicating that the received data input was held LOW for longer than a full-word transmission time (defined as start, data, parity, and stop bits). This bit is cleared to 0 after a write to UARTECR. In FIFO mode, this error is associated with the character at the top of the FIFO. When a break occurs, only one 0 character is loaded into the FIFO. The next character is only enabled after the receive data input goes to a 1 (marking state) and the next valid start bit is received.	WC	0x0
1	PE: Parity error. When set to 1, it indicates that the parity of the received data character does not match the parity that the EPS and SPS bits in the Line Control Register, UARTLCR_H. This bit is cleared to 0 by a write to UARTECR. In FIFO mode, this error is associated with the character at the top of the FIFO.	WC	0x0
0	FE: Framing error. When set to 1, it indicates that the received character did not have a valid stop bit (a valid stop bit is 1). This bit is cleared to 0 by a write to UARTECR. In FIFO mode, this error is associated with the character at the top of the FIFO.	WC	0x0

UART: UARTFR Register

Offset: 0x018

位	描述	类型	复位值
10	BE : 断帧错误。当检测到断帧条件，即接收数据输入线长时间保持低电平，超过完整字传输时间（包括起始位、数据位、校验位和停止位）时，该位置为1。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。断帧发生时，仅加载一个0字符入 FIFO。仅当接收数据输入恢复为高电平（标记状态），且接收到下一个有效起始位后，下一字符方被允许进入。	只读	-
9	PE : 校验错误。当该位为1时，表示接收的字符奇偶校验与 UARTECR_H（线路控制寄存器）中 EPS 和 SPS 位规定的不符。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。	只读	-
8	FE : 帧错误。当此位设置为1时，表示接收的字符没有有效停止位（有效停止位为1）。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关联。	只读	-
7:0	DATA : 接收（读取）数据字符。发送（写入）数据字符。	RWF	-

UART: UARTRSR 寄存器

偏移: 0x004

描述

接收状态寄存器/错误清除寄存器，UARTRSR/UARTECR

表427. UARTRSR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	OE : 溢出错误。若接收到数据且 FIFO 已满，该位被置为1。通过写入 UARTECR 可将该位清零。FIFO 内容保持有效，因为 FIFO 已满时不再写入新数据，仅覆盖移位寄存器的内容。CPU 必须读取数据以清空 FIFO。	WC	0x0
2	BE : 断帧错误。若检测到中断条件，该位被置为1，表示接收数据输入被保持为低电平，持续时间超过一个完整字的传输时间（定义为起始位、数据位、奇偶校验位和停止位）。该位在写入 UARTECR 后清零。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。发生 break 时，FIFO 中仅加载一个0字符。只有当接收数据输入变为1（标记状态）且接收到下一个有效起始位后，下一字符才被使能。	WC	0x0
1	PE : 校验错误。该位置1表示接收数据字符的奇偶校验与 UARTECR_H（线路控制寄存器）中的 EPS 和 SPS 位设置不符。通过写入 UARTECR 可将该位清零。 在 FIFO 模式下，该错误与 FIFO 顶部的字符相关联。	WC	0x0
0	FE : 帧错误。当此位设置为1时，表示接收的字符没有有效停止位（有效停止位为1）。该位通过写入 UARTECR 清零。在 FIFO 模式下，该错误与 FIFO 顶部的字符相关。	WC	0x0

UART: UARTFR 寄存器

偏移: 0x018

Description

Flag Register, UARTFR

Table 428. UARTFR Register

Bits	Description	Type	Reset
31:9	Reserved.	-	-
8	RI : Ring indicator. This bit is the complement of the UART ring indicator, nUARTRI, modem status input. That is, the bit is 1 when nUARTRI is LOW.	RO	-
7	TXFE : Transmit FIFO empty. The meaning of this bit depends on the state of the FEN bit in the Line Control Register, UARTLCR_H. If the FIFO is disabled, this bit is set when the transmit holding register is empty. If the FIFO is enabled, the TXFE bit is set when the transmit FIFO is empty. This bit does not indicate if there is data in the transmit shift register.	RO	0x1
6	RXFF : Receive FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H Register. If the FIFO is disabled, this bit is set when the receive holding register is full. If the FIFO is enabled, the RXFF bit is set when the receive FIFO is full.	RO	0x0
5	TXFF : Transmit FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H Register. If the FIFO is disabled, this bit is set when the transmit holding register is full. If the FIFO is enabled, the TXFF bit is set when the transmit FIFO is full.	RO	0x0
4	RXFE : Receive FIFO empty. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H Register. If the FIFO is disabled, this bit is set when the receive holding register is empty. If the FIFO is enabled, the RXFE bit is set when the receive FIFO is empty.	RO	0x1
3	BUSY : UART busy. If this bit is set to 1, the UART is busy transmitting data. This bit remains set until the complete byte, including all the stop bits, has been sent from the shift register. This bit is set as soon as the transmit FIFO becomes non-empty, regardless of whether the UART is enabled or not.	RO	0x0
2	DCD : Data carrier detect. This bit is the complement of the UART data carrier detect, nUARTDCD, modem status input. That is, the bit is 1 when nUARTDCD is LOW.	RO	-
1	DSR : Data set ready. This bit is the complement of the UART data set ready, nUARTDSR, modem status input. That is, the bit is 1 when nUARTDSR is LOW.	RO	-
0	CTS : Clear to send. This bit is the complement of the UART clear to send, nUARTCTS, modem status input. That is, the bit is 1 when nUARTCTS is LOW.	RO	-

UART: UARTEILPR Register

Offset: 0x020

Description

IrDA Low-Power Counter Register, UARTEILPR

Table 429. UARTEILPR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	ILPDVSR : 8-bit low-power divisor value. These bits are cleared to 0 at reset.	RW	0x00

UART: UARTEIBRD Register

Offset: 0x024

描述

标志寄存器，UARTFR

表428. UARTFR寄存器

位	描述	类型	复位值
31:9	保留。	-	-
8	RI : 环指示符。该位为 UART 环指示器 nUARTRI (调制解调器状态输入) 的反码。即当 nUARTRI 为 LOW 时, 该位为 1。	只读	-
7	TXFE : 发送 FIFO 为空。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当发送保持寄存器为空时该位被置位。若 FIFO 启用, 则当发送 FIFO 为空时 TXFE 位被置位。该位不指示发送移位寄存器中是否存在数据。	只读	0x1
6	RXFF : 接收 FIFO 已满。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当接收保持寄存器已满时该位被置位。若 FIFO 启用, 则当接收 FIFO 已满时 RXFF 位被置位。	只读	0x0
5	TXFF : 发送 FIFO 已满。该位的含义取决于 UARTLCR_H 寄存器中 FEN 位的状态。若 FIFO 被禁用, 则当发送保持寄存器已满时该位被置位。如果启用了 FIFO, 当传输FIFO满时, TXFF位将被置位。	只读	0x0
4	RXFE : 接收FIFO为空。该位的含义取决于UARTLCR_H寄存器中FEN位的状态。若FIFO被禁用, 当接收保持寄存器为空时, 该位将被置位。如果启用了FIFO, 当接收FIFO为空时, RXFE位将被置位。	只读	0x1
3	BUSY : UART 忙碌。当该位被置为 1 时, 表示 UART 正在忙于传输数据。该位将保持置位状态, 直至包含所有停止位的完整字节从移位寄存器发送完成。不论 UART 是否启用, 只要传输 FIFO 变为非空, 该位即被置位。	只读	0x0
2	DCD : 数据信号载波检测。该位为 UART 数据载波检测输入信号 nUARTDCD 的反码, 调制解调器状态输入。即当 nUARTDCD 为低电平时该位为 1。	只读	-
1	DSR : 数据集准备就绪。该位为 UART 数据集准备就绪信号 nUARTDSR 的反码, 调制解调器状态输入。即当 nUARTDSR 为低电平时, 该位为 1。	只读	-
0	CTS : 清除发送。该位为 UART 清除发送信号 nUARTCTS 的反码, 调制解调器状态输入。即当 nUARTCTS 为低电平时, 该位为 1。	只读	-

UART：UARTILPR寄存器

偏移: 0x020

描述

红外低功耗计数器寄存器，UARTILPR

表429. UARTILPR寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	ILPDVSR : 8位低功耗除数值。这些位在复位时全部清零。	读写	0x00

UART：UARTIBRD寄存器

偏移: 0x024

Description

Integer Baud Rate Register, UARTIBRD

Table 430. UARTIBRD Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	BAUD_DIVINT : The integer baud rate divisor. These bits are cleared to 0 on reset.	RW	0x0000

UART: UARTFBRD Register

Offset: 0x028

Description

Fractional Baud Rate Register, UARTFBRD

Table 431. UARTFBRD Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:0	BAUD_DIVFRAC : The fractional baud rate divisor. These bits are cleared to 0 on reset.	RW	0x00

UART: UARTLCR_H Register

Offset: 0x02c

Description

Line Control Register, UARTLCR_H

Table 432. UARTLCR_H Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	SPS : Stick parity select. 0 = stick parity is disabled 1 = either: * if the EPS bit is 0 then the parity bit is transmitted and checked as a 1 * if the EPS bit is 1 then the parity bit is transmitted and checked as a 0. This bit has no effect when the PEN bit disables parity checking and generation.	RW	0x0
6:5	WLEN : Word length. These bits indicate the number of data bits transmitted or received in a frame as follows: b11 = 8 bits b10 = 7 bits b01 = 6 bits b00 = 5 bits.	RW	0x0
4	FEN : Enable FIFOs: 0 = FIFOs are disabled (character mode) that is, the FIFOs become 1-byte-deep holding registers 1 = transmit and receive FIFO buffers are enabled (FIFO mode).	RW	0x0
3	STP2 : Two stop bits select. If this bit is set to 1, two stop bits are transmitted at the end of the frame. The receive logic does not check for two stop bits being received.	RW	0x0
2	EPS : Even parity select. Controls the type of parity the UART uses during transmission and reception: 0 = odd parity. The UART generates or checks for an odd number of 1s in the data and parity bits. 1 = even parity. The UART generates or checks for an even number of 1s in the data and parity bits. This bit has no effect when the PEN bit disables parity checking and generation.	RW	0x0
1	PEN : Parity enable: 0 = parity is disabled and no parity bit added to the data frame 1 = parity checking and generation is enabled.	RW	0x0

描述

整数波特率寄存器，UARTIBRD

表430. UARTIBRD寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	BAUD_DIVINT : 整数波特率除数。这些位在复位时全部清零。	读写	0x0000

UART：UARTFBRD寄存器

偏移：0x028

描述

分数波特率寄存器，UARTFBRD

表431. UARTFBRD寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:0	BAUD_DIVFRAC : 分数波特率除数。这些位在复位时全部清零。	读写	0x00

UART：UARTLCR_H 寄存器

偏移：0x02c

描述

线路控制寄存器，UARTLCR_H

表432。UARTLCR_H 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	SPS : 停用奇偶校验选择。0 = 停用奇偶校验；1 = 以下任一情况：* 当 EPS 位为0时，奇偶校验位以1传输和检测；* 当 EPS 位为1时，奇偶校验位以0传输和检测。当 PEN 位禁用奇偶校验的生成和检测时，此位无效。	读写	0x0
6:5	WLEN : 字长。这些位指示帧中传输或接收的数据位数，如下：b11 = 8位，b10 = 7位，b01 = 6位，b00 = 5位。	读写	0x0
4	FEN : 启用FIFO: 0 = 禁用FIFO（字符模式），即FIFO变为1字节深的保持寄存器；1 = 启用发送和接收FIFO缓冲区（FIFO模式）。	读写	0x0
3	STP2 : 选择两个停止位。如果此位设置为1，将在帧末尾发送两个停止位。接收逻辑不会检测是否接收了两个停止位。	读写	0x0
2	EPS : 奇偶校验选择。控制UART在发送和接收过程中使用的奇偶校验类型：0 = 奇校验。UART会生成或检测数据及校验位中的奇数个1。1 = 偶校验。UART会生成或检测数据及校验位中的偶数个1。当 PEN 位禁用奇偶校验的生成和检测时，此位无效。	读写	0x0
1	PEN : 奇偶校验使能；0 = 禁用奇偶校验且不在数据帧中添加校验位，1 = 启用奇偶校验的生成和检测。	读写	0x0

Bits	Description	Type	Reset
0	BRK: Send break. If this bit is set to 1, a low-level is continually output on the UARTRXD output, after completing transmission of the current character. For the proper execution of the break command, the software must set this bit for at least two complete frames. For normal use, this bit must be cleared to 0.	RW	0x0

UART: UARTCR Register

Offset: 0x030

Description

Control Register, UARTCR

Table 433. UARTCR Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15	CTSEN: CTS hardware flow control enable. If this bit is set to 1, CTS hardware flow control is enabled. Data is only transmitted when the nUARTCTS signal is asserted.	RW	0x0
14	RTSEN: RTS hardware flow control enable. If this bit is set to 1, RTS hardware flow control is enabled. Data is only requested when there is space in the receive FIFO for it to be received.	RW	0x0
13	OUT2: This bit is the complement of the UART Out2 (nUARTOut2) modem status output. That is, when the bit is programmed to a 1, the output is 0. For DTE this can be used as Ring Indicator (RI).	RW	0x0
12	OUT1: This bit is the complement of the UART Out1 (nUARTOut1) modem status output. That is, when the bit is programmed to a 1 the output is 0. For DTE this can be used as Data Carrier Detect (DCD).	RW	0x0
11	RTS: Request to send. This bit is the complement of the UART request to send, nUARTRTS, modem status output. That is, when the bit is programmed to a 1 then nUARTRTS is LOW.	RW	0x0
10	DTR: Data transmit ready. This bit is the complement of the UART data transmit ready, nUARTDTR, modem status output. That is, when the bit is programmed to a 1 then nUARTDTR is LOW.	RW	0x0
9	RXE: Receive enable. If this bit is set to 1, the receive section of the UART is enabled. Data reception occurs for either UART signals or SIR signals depending on the setting of the SIREN bit. When the UART is disabled in the middle of reception, it completes the current character before stopping.	RW	0x1
8	TXE: Transmit enable. If this bit is set to 1, the transmit section of the UART is enabled. Data transmission occurs for either UART signals, or SIR signals depending on the setting of the SIREN bit. When the UART is disabled in the middle of transmission, it completes the current character before stopping.	RW	0x1

位	描述	类型	复位值
0	BRK : 发送中断信号。如果此位设置为1，则在完成当前字符的传输后，UART TTXD输出端持续输出低电平。为正确执行中断命令，软件必须将此位至少保持两个完整帧的时间。在正常使用情况下，该位必须清零（置为0）。	读写	0x0

UART: UARTCR 寄存器

偏移: 0x030

描述

控制寄存器，UARTCR

表 433. UARTCR 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	CTSEN : CTS 硬件流控制使能。当该位被置为1时，启用CTS硬件流控制。仅当nUARTCTS信号被断言时，才发送数据。	读写	0x0
14	RTSEN : RTS 硬件流控制使能。当该位被置为1时，启用RTS硬件流控制。仅当接收FIFO有空间时，才请求数据接收。	读写	0x0
13	OUT2 : 该位为UART Out2 (nUARTOut2) 调制解调器状态输出的反码。即当该位被设置为1时，输出值为0。对于DTE，该位可用作环指示 (RI)。	读写	0x0
12	OUT1 : 该位为UART Out1 (nUARTOut1) 调制解调器状态输出的反码。亦即，当该位被置为1时，输出为0。对于DTE，该位可用作数据载波检测 (DCD)。	读写	0x0
11	RTS : 请求发送。该位为UART请求发送信号nUARTRTS (调制解调器状态输出) 之反码。亦即，当该位被置为1时，nUARTRTS为低电平。	读写	0x0
10	DTR : 数据传输准备。该位为UART数据传输准备信号nUARTDTR (调制解调器状态输出) 之反码。亦即，当该位被置为1时，nUARTDTR为低电平。	读写	0x0
9	RXE : 接收使能。当该位被置为1时，UART接收部分被启用。数据接收可基于SIREN位设置，接收UART信号或SIR信号。当UART在接收过程中被禁用时，会完成当前字符的接收后再停止。	读写	0x1
8	TXE : 发送使能。若此位被置为1，则UART的发送部分被使能。数据传输根据SIREN位的设置，通过UART信号或SIR信号进行。若UART在传输过程中被禁用，则会完成当前字符传输后停止。	读写	0x1

Bits	Description	Type	Reset
7	LBE: Loopback enable. If this bit is set to 1 and the SIREN bit is set to 1 and the SIRTEST bit in the Test Control Register, UARTTCR is set to 1, then the nSIROUT path is inverted, and fed through to the SIRIN path. The SIRTEST bit in the test register must be set to 1 to override the normal half-duplex SIR operation. This must be the requirement for accessing the test registers during normal operation, and SIRTEST must be cleared to 0 when loopback testing is finished. This feature reduces the amount of external coupling required during system test. If this bit is set to 1, and the SIRTEST bit is set to 0, the UARTRXD path is fed through to the UARTRXD path. In either SIR mode or UART mode, when this bit is set, the modem outputs are also fed through to the modem inputs. This bit is cleared to 0 on reset, to disable loopback.	RW	0x0
6:3	Reserved.	-	-
2	SIRLP: SIR low-power IrDA mode. This bit selects the IrDA encoding mode. If this bit is cleared to 0, low-level bits are transmitted as an active high pulse with a width of 3 / 16th of the bit period. If this bit is set to 1, low-level bits are transmitted with a pulse width that is 3 times the period of the IrLPBaud16 input signal, regardless of the selected bit rate. Setting this bit uses less power, but might reduce transmission distances.	RW	0x0
1	SIREN: SIR enable: 0 = IrDA SIR ENDEC is disabled. nSIROUT remains LOW (no light pulse generated), and signal transitions on SIRIN have no effect. 1 = IrDA SIR ENDEC is enabled. Data is transmitted and received on nSIROUT and SIRIN. UARTRXD remains HIGH, in the marking state. Signal transitions on UARTRXD or modem status inputs have no effect. This bit has no effect if the UARTEN bit disables the UART.	RW	0x0
0	UARTEN: UART enable: 0 = UART is disabled. If the UART is disabled in the middle of transmission or reception, it completes the current character before stopping. 1 = the UART is enabled. Data transmission and reception occurs for either UART signals or SIR signals depending on the setting of the SIREN bit.	RW	0x0

UART: UARTIFLS Register

Offset: 0x034

Description

Interrupt FIFO Level Select Register, UARTIFLS

Table 434. UARTIFLS Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5:3	RXIFLSEL: Receive interrupt FIFO level select. The trigger points for the receive interrupt are as follows: b000 = Receive FIFO becomes \geq 1 / 8 full b001 = Receive FIFO becomes \geq 1 / 4 full b010 = Receive FIFO becomes \geq 1 / 2 full b011 = Receive FIFO becomes \geq 3 / 4 full b100 = Receive FIFO becomes \geq 7 / 8 full b101-b111 = reserved.	RW	0x2
2:0	TXIFLSEL: Transmit interrupt FIFO level select. The trigger points for the transmit interrupt are as follows: b000 = Transmit FIFO becomes \leq 1 / 8 full b001 = Transmit FIFO becomes \leq 1 / 4 full b010 = Transmit FIFO becomes \leq 1 / 2 full b011 = Transmit FIFO becomes \leq 3 / 4 full b100 = Transmit FIFO becomes \leq 7 / 8 full b101-b111 = reserved.	RW	0x2

UART: UARTIMSC Register

位	描述	类型	复位值
7	LBE : 回环使能。若此位设置为1，且SIREN位为1，同时测试控制寄存器UARTTCR中的SIRTEST位为1，则nSIROUT路径将被反相，并反馈至SIRIN路径。测试寄存器中的SIRTEST位必须设置为1，方可覆盖正常的半双工SIR操作。此为正常操作期间访问测试寄存器的必要条件，回环测试结束后，须将SIRTEST清零。该功能减少了系统测试期间所需的外部耦合量。如果该位被设置为1，且SIRTEST位被设置为0，则UARTTXD路径会传递至UARTRXD路径。无论在SIR模式还是UART模式，当该位被设置时，调制解调器输出同样会传递至调制解调器输入。该位在复位时被清零（设置为0），以禁用环回功能。	读写	0x0
6:3	保留。	-	-
2	SIRLP : 低功耗IrDA SIR模式。该位用于选择IrDA编码模式。如果该位被清零（设置为0），则低电平比特以宽度为比特周期3/16的高电平脉冲传输。如果该位被设置为1，低电平比特的脉冲宽度为IrLPBaud16输入信号周期的3倍，与所选比特率无关。设置该位可降低功耗，但可能会缩短传输距离。	读写	0x0
1	SIREN : SIR使能：0 = 禁用IrDA SIR ENDEC。nSIROUT保持低电平（未产生光脉冲），SIRIN上的信号跳变无效。1 = 启用IrDA SIR 编解码器。数据通过nSIROUT和SIRIN进行传输和接收。UARTTXD保持高电平，处于标记状态。UARTRXD或调制解调器状态输入端的信号跳变无效。如果UARTEN位禁用UART，则此位无效。	读写	0x0
0	UARTEN : UART使能，0 = UART已禁用。如果在传输或接收过程中禁用UART，则会完成当前字符传输后停止。1 = UART已启用。数据传输和接收根据SIREN位设置，采用UART信号或SIR信号。	读写	0x0

UART: UARTIFLS 寄存器

偏移: 0x034

描述

中断 FIFO 水平选择寄存器，UARTIFLS

表 434. UARTIFLS 寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5:3	RXIFLSEL : 接收中断 FIFO 水平选择。接收中断触发点如下：b000 = 接收 FIFO 达到或超过 1/8 满；b001 = 接收 FIFO 达到或超过 1/4 满；b010 = 接收 FIFO 达到或超过 1/2 满。 b011 = 接收 FIFO 满 >= 3/4 b100 = 接收 FIFO 满 >= 7/8 b101-b111 = 保留。	读写	0x2
2:0	TXIFLSEL : 发送中断 FIFO 水平选择。发送中断的触发点如下：b000 = 发送 FIFO 满 <= 1/8 b001 = 发送 FIFO 满 <= 1/4 b010 = 发送 FIFO 满 <= 1/2 b011 = 发送 FIFO 满 <= 3/4 b100 = 发送 FIFO 满 <= 7/8 b101-b111 = 保留。	读写	0x2

UART: UARTIMSC 寄存器

Offset: 0x038**Description**

Interrupt Mask Set/Clear Register, UARTIMSC

Table 435. UARTIMSC Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-
10	OEIM : Overrun error interrupt mask. A read returns the current mask for the UARTOEINTR interrupt. On a write of 1, the mask of the UARTOEINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
9	BEIM : Break error interrupt mask. A read returns the current mask for the UARTBEINTR interrupt. On a write of 1, the mask of the UARTBEINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
8	PEIM : Parity error interrupt mask. A read returns the current mask for the UARTPEINTR interrupt. On a write of 1, the mask of the UARTPEINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
7	FEIM : Framing error interrupt mask. A read returns the current mask for the UARTFEINTR interrupt. On a write of 1, the mask of the UARTFEINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
6	RTIM : Receive timeout interrupt mask. A read returns the current mask for the UARTRTINTR interrupt. On a write of 1, the mask of the UARTRTINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
5	TXIM : Transmit interrupt mask. A read returns the current mask for the UARTRXINTR interrupt. On a write of 1, the mask of the UARTRXINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
4	RXIM : Receive interrupt mask. A read returns the current mask for the UARTRXINTR interrupt. On a write of 1, the mask of the UARTRXINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
3	DSRMIM : nUARTDSR modem interrupt mask. A read returns the current mask for the UARDSRINTR interrupt. On a write of 1, the mask of the UARDSRINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
2	DCDMIM : nUARTDCD modem interrupt mask. A read returns the current mask for the UARDCDINTR interrupt. On a write of 1, the mask of the UARDCDINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
1	CTSMIM : nUARTCTS modem interrupt mask. A read returns the current mask for the UARTCTSINTR interrupt. On a write of 1, the mask of the UARTCTSINTR interrupt is set. A write of 0 clears the mask.	RW	0x0
0	RIMIM : nUARTRI modem interrupt mask. A read returns the current mask for the UARTRIINTR interrupt. On a write of 1, the mask of the UARTRIINTR interrupt is set. A write of 0 clears the mask.	RW	0x0

UART: UARTRIS Register**Offset:** 0x03c**Description**

Raw Interrupt Status Register, UARTRIS

Table 436. UARTRIS Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-

偏移: 0x038

说明

中断屏蔽设置/清除寄存器, UARTIMSC

表 435. UARTIMSC 寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEIM: 溢出错误中断屏蔽。读取操作返回当前 UARTOEINTR 中断的屏蔽状态。写入 1 时, 设置 UARTOEINTR 中断屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
9	BEIM: 中断错误中断掩码。读取返回UARTBEINTR中断的当前掩码。写入1时, UARTBEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
8	PEIM: 奇偶校验错误中断掩码。读取返回UARTPEINTR中断的当前掩码。写入1时, UARTPEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
7	FEIM: 帧错误中断掩码。读取返回UARTFEINTR中断的当前掩码。写入1时, UARTFEINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
6	RTIM: 接收超时中断掩码。读取返回UARTRTINTR中断的当前掩码。写入1时, UARTRTINTR中断掩码被设置; 写入0时, 掩码被清除。	读写	0x0
5	TXIM: 发送中断屏蔽。读取操作返回 UARTRXINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTRXINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
4	RXIM: 接收中断屏蔽。读取操作返回 UARTRXINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTRXINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
3	DSRMIM: nUARTDSR 调制解调器中断屏蔽。读取操作返回 UARTDSRINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTDSRINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
2	DCDMIM: nUARTDCD 调制解调器中断屏蔽。读取操作返回 UARTDCDINTR 中断的当前屏蔽状态。写入 1 时, 设置 UARTDCDINTR 中断的屏蔽; 写入 0 时, 清除该屏蔽。	读写	0x0
1	CTSMIM: nUARTCTS 调制解调器中断屏蔽。读取返回 UARTCTSINTR 中断的当前屏蔽状态。写入1时, 设置 UARTCTSINTR 中断屏蔽; 写入0时, 清除该屏蔽。	读写	0x0
0	RIMIM: nUARTRI 调制解调器中断屏蔽。读取返回 UARTRIINTR 中断的当前屏蔽状态。写入1时, 设置 UARTRIINTR 中断屏蔽; 写入0时, 清除该屏蔽。	读写	0x0

UART: UARTRIS 寄存器

偏移量: 0x03c

描述

原始中断状态寄存器, UARTRIS

表436. UARTRIS 寄存器

位	描述	类型	复位值
31:11	保留。	-	-

Bits	Description	Type	Reset
10	OERIS: Overrun error interrupt status. Returns the raw interrupt state of the UARTOEINTR interrupt.	RO	0x0
9	BERIS: Break error interrupt status. Returns the raw interrupt state of the UARTBEINTR interrupt.	RO	0x0
8	PERIS: Parity error interrupt status. Returns the raw interrupt state of the UARTPEINTR interrupt.	RO	0x0
7	FERIS: Framing error interrupt status. Returns the raw interrupt state of the UARTFEINTR interrupt.	RO	0x0
6	RTRIS: Receive timeout interrupt status. Returns the raw interrupt state of the UARTRTINTR interrupt. a	RO	0x0
5	TXRIS: Transmit interrupt status. Returns the raw interrupt state of the UARTTXINTR interrupt.	RO	0x0
4	RXRIS: Receive interrupt status. Returns the raw interrupt state of the UARTRXINTR interrupt.	RO	0x0
3	DSRRMIS: nUARTDSR modem interrupt status. Returns the raw interrupt state of the UARTDSRINTR interrupt.	RO	-
2	DCDRMIS: nUARTDCD modem interrupt status. Returns the raw interrupt state of the UARTDCDINTR interrupt.	RO	-
1	CTSRMIS: nUARTCTS modem interrupt status. Returns the raw interrupt state of the UARTCTSINTR interrupt.	RO	-
0	RIRMIS: nUARTRI modem interrupt status. Returns the raw interrupt state of the UARTRIINTR interrupt.	RO	-

UART: UARTMIS Register

Offset: 0x040

Description

Masked Interrupt Status Register, UARTMIS

Table 437. UARTMIS Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-
10	OEMIS: Overrun error masked interrupt status. Returns the masked interrupt state of the UARTOEINTR interrupt.	RO	0x0
9	BEMIS: Break error masked interrupt status. Returns the masked interrupt state of the UARTBEINTR interrupt.	RO	0x0
8	PEMIS: Parity error masked interrupt status. Returns the masked interrupt state of the UARTPEINTR interrupt.	RO	0x0
7	FEMIS: Framing error masked interrupt status. Returns the masked interrupt state of the UARTFEINTR interrupt.	RO	0x0
6	RTMIS: Receive timeout masked interrupt status. Returns the masked interrupt state of the UARTRTINTR interrupt.	RO	0x0
5	TXMIS: Transmit masked interrupt status. Returns the masked interrupt state of the UARTTXINTR interrupt.	RO	0x0

位	描述	类型	复位值
10	OERIS: 溢出错误中断状态。返回UARTOEINTR中断的原始中断状态。	只读	0x0
9	BERIS: 断开错误中断状态。返回UARTBEINTR中断的原始中断状态。	只读	0x0
8	PERIS: 奇偶校验错误中断状态。返回UARTPEINTR中断的原始中断状态。 ◦	只读	0x0
7	FERIS: 帧错误中断状态。返回UARTFEINTR中断的原始中断状态。	只读	0x0
6	RTRIS: 接收超时中断状态。返回UARTRTINTR中断的原始中断状态。	只读	0x0
5	TXRIS: 发送中断状态。返回UARTTXINTR中断的原始中断状态。	只读	0x0
4	RXRIS: 接收中断状态。返回UARTRXINTR中断的原始中断状态。	只读	0x0
3	DSRRMIS: nUARTDSR调制解调器中断状态。返回UARTDSRINTR中断的原始中断状态。	只读	-
2	DCDRMIS: nUARTDCD调制解调器中断状态。返回UARTDCDINTR中断的原始中断状态。	只读	-
1	CTSRMIS: nUARTCTS调制解调器中断状态。返回UARTCTSINTR中断的原始中断状态。	只读	-
0	RIRMIS: nUARTRI调制解调器中断状态。返回UARTRIINTR中断的原始中断状态。	只读	-

UART：UARTMIS寄存器

偏移量: 0x040

描述

掩码中断状态寄存器，UARTMIS

表437. UARTMIS寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEMIS: 溢出错误屏蔽中断状态。返回UARTOEINTR中断的屏蔽中断状态。	只读	0x0
9	BEMIS: 断点错误屏蔽中断状态。返回UARTBEINTR中断的屏蔽中断状态。 ◦	只读	0x0
8	PEMIS: 奇偶校验错误屏蔽中断状态。返回UARTPEINTR中断的屏蔽中断状态。	只读	0x0
7	FEMIS: 帧错误屏蔽中断状态。返回UARTFEINTR中断的屏蔽中断状态。	只读	0x0
6	RTMIS: 接收超时屏蔽中断状态。返回UARTRTINTR中断的屏蔽中断状态。	只读	0x0
5	TXMIS: 发送屏蔽中断状态。返回UARTTXINTR中断的屏蔽中断状态。	只读	0x0

Bits	Description	Type	Reset
4	RXMISS: Receive masked interrupt status. Returns the masked interrupt state of the UARTRXINTR interrupt.	RO	0x0
3	DSRMMIS: nUARTDSR modem masked interrupt status. Returns the masked interrupt state of the UARTDSRINTR interrupt.	RO	-
2	DCDMMIS: nUARTDCD modem masked interrupt status. Returns the masked interrupt state of the UARTDCDINTR interrupt.	RO	-
1	CTSMMIS: nUARTCTS modem masked interrupt status. Returns the masked interrupt state of the UARTCTSINTR interrupt.	RO	-
0	RIMMIS: nUARTRI modem masked interrupt status. Returns the masked interrupt state of the UARTRIINTR interrupt.	RO	-

UART: UARTICR Register

Offset: 0x044

Description

Interrupt Clear Register, UARTICR

Table 438. UARTICR Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-
10	OEIC: Overrun error interrupt clear. Clears the UARTOEINTR interrupt.	WC	-
9	BEIC: Break error interrupt clear. Clears the UARTBEINTR interrupt.	WC	-
8	PEIC: Parity error interrupt clear. Clears the UARTPEINTR interrupt.	WC	-
7	FEIC: Framing error interrupt clear. Clears the UARTFEINTR interrupt.	WC	-
6	RTIC: Receive timeout interrupt clear. Clears the UARTRTINTR interrupt.	WC	-
5	TXIC: Transmit interrupt clear. Clears the UARTRXINTR interrupt.	WC	-
4	RXIC: Receive interrupt clear. Clears the UARTRXINTR interrupt.	WC	-
3	DSRMIC: nUARTDSR modem interrupt clear. Clears the UARTDSRINTR interrupt.	WC	-
2	DCDMIC: nUARTDCD modem interrupt clear. Clears the UARTDCDINTR interrupt.	WC	-
1	CTSMIC: nUARTCTS modem interrupt clear. Clears the UARTCTSINTR interrupt.	WC	-
0	RIMIC: nUARTRI modem interrupt clear. Clears the UARTRIINTR interrupt.	WC	-

UART: UARTRDMACR Register

Offset: 0x048

Description

DMA Control Register, UARTRDMACR

Table 439.
UARTRDMACR Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-

位	描述	类型	复位值
4	RXMIS : 接收屏蔽中断状态。返回UARTRXINTR中断的屏蔽中断状态。	只读	0x0
3	DSRMMIS : nUARTDSR调制解调器屏蔽中断状态。返回UARTDSRINTR中断的屏蔽中断状态。	只读	-
2	DCDMMIS : nUARTDCD调制解调器屏蔽中断状态。返回UARTDCDINTR中断的屏蔽中断状态。	只读	-
1	CTSMMIS : nUARTCTS调制解调器屏蔽中断状态。返回UARTCTSINTR中断的屏蔽中断状态。	只读	-
0	RIMMIS : nUARTRI调制解调器屏蔽中断状态。返回UARTRIINTR中断的屏蔽中断状态。	只读	-

UART: UARTICR寄存器

偏移量: 0x044

描述

中断清除寄存器, UARTICR

表438. UARTICR寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	OEIC : 溢出错误中断清除。清除UARTOEINTR中断。	WC	-
9	BEIC : 断点错误中断清除。清除UARTBEINTR中断。	WC	-
8	PEIC : 奇偶校验错误中断清除。清除UARTPEINTR中断。	WC	-
7	FEIC : 帧错误中断清除。清除UARTFEINTR中断。	WC	-
6	RTIC : 接收超时中断清除。清除UARTRTINTR中断。	WC	-
5	TXIC : 发送中断清除。清除UARTTXINTR中断。	WC	-
4	RXIC : 接收中断清除。清除UARTRXINTR中断。	WC	-
3	DSRMIC : nUARTDSR 调制解调器中断清除。清除UARTDSRINTR中断 。	WC	-
2	DCDMIC : nUARTDCD 调制解调器中断清除。清除UARTDCDINTR中断 。	WC	-
1	CTSMIC : nUARTCTS 调制解调器中断清除。清除UARTCTSINTR中断 。	WC	-
0	RIMIC : nUARTRI 调制解调器中断清除。清除UARTRIINTR中断。	WC	-

UART: UARTDMACR 寄存器

偏移: 0x048

描述

DMA 控制寄存器, UARTDMACR

表 439。
UARTDMACR 寄存器

位	描述	类型	复位值
31:3	保留。	-	-

Bits	Description	Type	Reset
2	DMAONERR: DMA on error. If this bit is set to 1, the DMA receive request outputs, UARTRXDMASREQ or UARTRXDMABREQ, are disabled when the UART error interrupt is asserted.	RW	0x0
1	TXDMAE: Transmit DMA enable. If this bit is set to 1, DMA for the transmit FIFO is enabled.	RW	0x0
0	RXDMAE: Receive DMA enable. If this bit is set to 1, DMA for the receive FIFO is enabled.	RW	0x0

UART: UARTPERIPHID0 Register

Offset: 0xfe0

Description

UARTPeriphID0 Register

Table 440.
UARTPERIPHID0
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	PARTNUMBER0: These bits read back as 0x11	RO	0x11

UART: UARTPERIPHID1 Register

Offset: 0xfe4

Description

UARTPeriphID1 Register

Table 441.
UARTPERIPHID1
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:4	DESIGNERO: These bits read back as 0x1	RO	0x1
3:0	PARTNUMBER1: These bits read back as 0x0	RO	0x0

UART: UARTPERIPHID2 Register

Offset: 0xfe8

Description

UARTPeriphID2 Register

Table 442.
UARTPERIPHID2
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:4	REVISION: This field depends on the revision of the UART: r1p0 0x0 r1p1 0x1 r1p3 0x2 r1p4 0x2 r1p5 0x3	RO	0x3
3:0	DESIGNER1: These bits read back as 0x4	RO	0x4

UART: UARTPERIPHID3 Register

Offset: 0xfc

Description

UARTPeriphID3 Register

位	描述	类型	复位值
2	DMAONERR : DMA错误时。若此位被置为1，当UART错误中断发生时，UART接收请求输出UARTRXDMASREQ或UARTRXDMABREQ将被禁用。	读写	0x0
1	TXDMAE : 传输DMA使能。若此位被置为1，则启用传输FIFO的DMA功能。 ◦	读写	0x0
0	RXDMAE : 接收DMA使能。若此位被置为1，则启用接收FIFO的DMA功能。	读写	0x0

UART：UARTPERIPHID0 寄存器

偏移量：0xfe0

描述

UARTPeriphID0 寄存器

表440。
UARTPERIPHID0
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	PARTNUMBER0 : 这些位的读出值为0x11	只读	0x11

UART：UARTPERIPHID1 寄存器

偏移量：0xfe4

描述

UARTPeriphID1 寄存器

表441。
UARTPERIPHID1
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	DESIGNERO : 这些位的读出值为0x1	只读	0x1
3:0	PARTNUMBER1 : 这些位读取值为 0x0	只读	0x0

UART：UARTPERIPHID2 寄存器

偏移量：0xfe8

描述

UARTPeriphID2 寄存器

表442。
UARTPERIPHID2
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	版本号：该字段取决于UART的版本：r1p0 0x0, r1p1 0x1 r1p3 0x2, r1p4 0x2, r1p5 0x3	只读	0x3
3:0	DESIGNER1 : 这些位读取值为 0x4	只读	0x4

UART：UARTPERIPHID3 寄存器

偏移量：0fec

描述

UARTPeriphID3 寄存器

Table 443.
UARTPERIPHID3
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	CONFIGURATION: These bits read back as 0x00	RO	0x00

UART: UARTPCELLID0 Register

Offset: 0xff0

Description

UARTPCELLID0 Register

Table 444.
UARTPCELLID0
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	UARTPCELLID0: These bits read back as 0x0D	RO	0x0d

UART: UARTPCELLID1 Register

Offset: 0xff4

Description

UARTPCELLID1 Register

Table 445.
UARTPCELLID1
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	UARTPCELLID1: These bits read back as 0xF0	RO	0xf0

UART: UARTPCELLID2 Register

Offset: 0xff8

Description

UARTPCELLID2 Register

Table 446.
UARTPCELLID2
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	UARTPCELLID2: These bits read back as 0x05	RO	0x05

UART: UARTPCELLID3 Register

Offset: 0ffc

Description

UARTPCELLID3 Register

Table 447.
UARTPCELLID3
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	UARTPCELLID3: These bits read back as 0xB1	RO	0xb1

表 443。
UARTPERIPHID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	配置：这些位读取值为 0x00	只读	0x00

UART：UARTPCELLID0 寄存器

偏移量：0xff0

描述

UARTPCellID0 寄存器

表 444。
UARTPCELLID0
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID0：这些位读取的返回值为 0x0D	只读	0x0d

UART：UARTPCELLID1 寄存器

偏移：0xff4

描述

UARTPCellID1 寄存器

表 445。
UARTPCELLID1
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID1：这些位读取的返回值为 0xF0	只读	0xf0

UART：UARTPCELLID2 寄存器

偏移：0xff8

描述

UARTPCellID2 寄存器

表 446。
UARTPCELLID2
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID2：这些位读取的返回值为 0x05	只读	0x05

UART：UARTPCELLID3 寄存器

偏移：0ffc

描述

UARTPCellID3 寄存器

表 447。
UARTPCELLID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	UARTPCELLID3：这些位读取的返回值为 0xB1	只读	0xb1

4.3. I2C

Synopsys Documentation

Synopsys Proprietary. Used with permission.

I2C is a commonly used 2-wire interface that can be used to connect devices for low speed data transfer using clock `SCL` and data `SDA` wires.

RP2040 has two identical instances of an I2C controller. The external pins of each controller are connected to GPIO pins as defined in the GPIO muxing [table](#) in [Section 2.19.2](#). The muxing options give some IO flexibility.

4.3.1. Features

Each I2C controller is based on a configuration of the Synopsys DW_apb_i2c (v2.01) IP. The following features are supported:

- Master or Slave (Default to Master mode)
- Standard mode, Fast mode or Fast mode plus
- Default slave address 0x055
- Supports 10-bit addressing in Master mode
- 16-element transmit buffer
- 16-element receive buffer
- Can be driven from DMA
- Can generate interrupts

4.3.1.1. Standard

The I2C controller was designed for I2C Bus specification, version 6.0, dated April 2014.

4.3.1.2. Clocking

All clocks in the I2C controller are connected to `clk_sys`, including `ic_clk` which is mentioned in later sections. The I2C clock is generated by dividing down this clock, controlled by registers inside the block.

4.3.1.3. IOs

Each controller must connect its clock `SCL` and data `SDA` to one pair of GPIOs. The I2C standard requires that drivers drive a signal low, or when not driven the signal will be pulled high. This applies to SCL and SDA. The GPIO pads should be configured for:

- pull-up enabled
- slew rate limited
- schmitt trigger enabled

4.3. I2C

Synopsys文档

Synopsys专有，经过授权使用。

I2C是一种常用的两线接口，通过时钟线 **SCL**和数据线 **SDA**连接设备，用于低速数据传输。

RP2040包含两个相同的I2C控制器实例。每个控制器的外部引脚连接至第2.19.2节GPIO复用表中定义的GPIO引脚，该复用选项提供了一定的输入输出灵活性。

4.3.1. 特性

每个I2C控制器基于Synopsys DW_apb_i2c (v2.01) IP配置，支持以下功能：

- 主设备模式或从设备模式（默认为主设备模式）
- 支持标准模式、快速模式及快速增强模式
- 默认从设备地址为0x055
- 主设备模式支持10位地址
- 16单元传输缓存
- 16单元接收缓存
- 支持由DMA驱动
- 支持中断生成

4.3.1.1. 标准

I2C控制器依据2014年4月发布的I2C总线规范6.0版设计。

4.3.1.2. 时钟

I2C控制器的所有时钟均连接至 **clk_sys**，包括后文所述的 **ic_clk**。I2C时钟通过分频该时钟产生，由模块内部寄存器控制。

4.3.1.3. 输入/输出

每个控制器必须将其时钟线 **SCL**和数据线 **SDA**连接至一对GPIO。I2C标准要求驱动器主动拉低信号，若未驱动信号则自动被上拉。此要求适用于SCL和SDA。GPIO引脚应配置为：

- 启用上拉
- 限制转换速率
- 启用施密特触发器

NOTE

There should also be external pull-ups on the board as the internal pad pull-ups may not be strong enough to pull up external circuits.

4.3.2. IP Configuration

I2C configuration details (each instance is fully independent):

- 32-bit APB access
- Supports Standard mode, Fast mode or Fast mode plus (not High speed)
- Default slave address of 0x055
- Master or Slave mode
- Master by default (Slave mode disabled at reset)
- 10-bit addressing supported in master mode (7-bit by default)
- 16 entry transmit buffer
- 16 entry receive buffer
- Allows restart conditions when a master (can be disabled for legacy device support)
- Configurable timing to adjust TsuDAT/ThDAT
- General calls responded to on reset
- Interface to DMA
- Single interrupt output
- Configurable timing to adjust clock frequency
- Spike suppression (default 7 clk_sys cycles)
- Can NACK after data received by Slave
- Hold transfer when TX FIFO empty
- Hold bus until space available in RX FIFO
- Restart detect interrupt in Slave mode
- Optional blocking Master commands (not enabled by default)

4.3.3. I2C Overview

The I2C bus is a 2-wire serial interface, consisting of a serial data line **SDA** and a serial clock **SCL**. These wires carry information between the devices connected to the bus. Each device is recognized by a unique address and can operate as either a “transmitter” or “receiver”, depending on the function of the device. Devices can also be considered as masters or slaves when performing data transfers. A master is a device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.

● 注意

板上应配备外部上拉电阻，因为内部引脚的上拉能力可能不足以满足外部电路的上拉需求。

4.3.2. I₂C 配置

I₂C 配置细节（每个实例完全独立）：

- 32 位 APB 访问
- 支持标准模式、快速模式或快速模式增强版（不支持高速模式）
- 默认从机地址为 0x055
- 主模式或从模式
- 默认主模式（复位时从模式禁用）
- 主模式支持 10 位寻址（默认 7 位）
- 16 项传输缓冲区
- 16 项接收缓冲区
- 允许主机发起重启条件（可为兼容旧设备而禁用）
- 可配置时序以调整 TsuDAT/ThDAT
- 复位时响应广播调用
- DMA 接口
- 单一中断输出
- 可配置时序以调整时钟频率
- 尖峰抑制（默认 7 个 clk_sys 周期）
- 从机接收数据后可发送 NACK
- 当 TX FIFO 为空时保持传输
- 保持总线，直到 RX FIFO 中有可用空间
- 从模式下重新检测中断
- 可选阻塞式主机命令（默认未启用）

4.3.3. I₂C 概述

I₂C 总线是一种由串行数据线 SDA 和串行时钟线 SCL 组成的双线串行接口。这些线路在总线上连接的设备之间传递信息。每个设备以唯一地址识别，且可根据设备功能作为“发送方”或“接收方”操作。设备在执行数据传输时，也可以被视为主设备或从设备。主设备是指启动总线数据传输并生成时钟信号以允许该传输的设备。此时，任何被寻址的设备均视为从设备。

NOTE

The I2C block must only be programmed to operate in either master OR slave mode only. Operating as a master and slave simultaneously is not supported.

The I2C block can operate in these modes:

- standard mode (with data rates from 0 to 100kbps),
- fast mode (with data rates less than or equal to 400kbps),
- fast mode plus (with data rates less than or equal to 1000kbps).

These modes are not supported:

- High-speed mode (with data rates less than or equal to 3.4Mbps),
- Ultra-Fast Speed Mode (with data rates less than or equal to 5Mbps).

NOTE

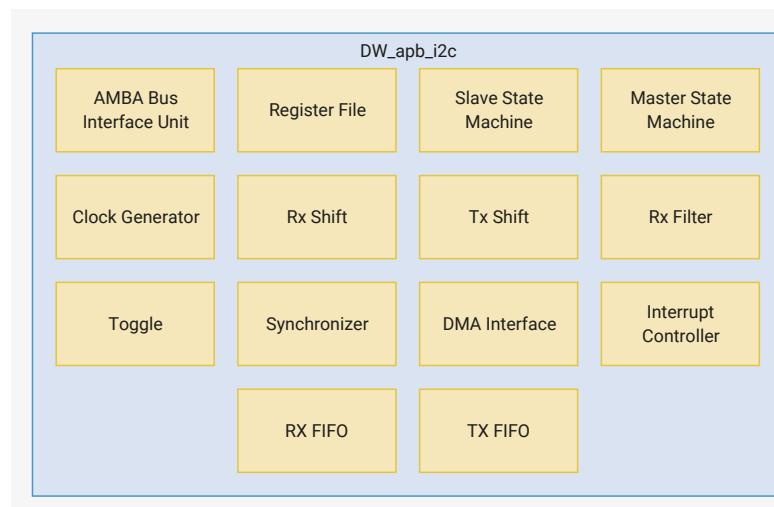
References to fast mode also apply to fast mode plus, unless specifically stated otherwise.

The I2C block can communicate with devices in one of these modes as long as they are attached to the bus. Additionally, fast mode devices are downward compatible. For instance, fast mode devices can communicate with standard mode devices in 0 to 100kbps I2C bus system. However standard mode devices are not upward compatible and should not be incorporated in a fast-mode I2C bus system as they cannot follow the higher transfer rate and unpredictable states would occur.

An example of high-speed mode devices are LCD displays, high-bit count ADCs, and high capacity EEPROMs. These devices typically need to transfer large amounts of data. Most maintenance and control applications, the common use for the I2C bus, typically operate at 100kHz (in standard and fast modes). Any DW_apb_i2c device can be attached to an I2C-bus and every device can talk with any master, passing information back and forth. There needs to be at least one master (such as a microcontroller or DSP) on the bus but there can be multiple masters, which require them to arbitrate for ownership. Multiple masters and arbitration are explained later in this chapter. The I2C block does not support SMBus and PMBus protocols (for System Management and Power management).

The DW_apb_i2c is made up of an AMBA APB slave interface, an I2C interface, and FIFO logic to maintain coherency between the two interfaces. The blocks of the component are illustrated in [Figure 64](#).

Figure 64. I2C Block diagram



The following define the functions of the blocks in [Figure 64](#):

- **AMBA Bus Interface Unit** – Takes the APB interface signals and translates them into a common generic interface that allows the register file to be bus protocol-agnostic.

注意

I2C模块必须仅被编程为主模式或从模式中的一种。不支持主从并存的操作模式。

I2C模块可在以下模式下运行：

- 标准模式（数据速率范围0至100kbps），
- 快速模式（数据传输速率小于或等于400kbps），
- 快速增强模式（数据传输速率小于或等于1000kbps）。

以下模式不受支持：

- 高速模式（数据传输速率小于或等于3.4Mbps），
- 超高速模式（数据传输速率小于或等于5Mbps）。

注意

对快速模式的提及同样适用于快速增强模式，除非另有明确说明。

只要设备连接到总线，I2C模块即可与这些模式的设备通信。

此外，快速模式设备具有向下兼容性。例如，快速模式设备能够在0至100kbps的I2C总线系统中与标准模式设备通信。然而，标准模式设备不具备向上兼容性，不应纳入快速模式I2C总线系统中，因为其无法适应更高的传输速率，可能导致不可预测的状态。

高速模式设备的示例包括液晶显示器、高位数模数转换器和大容量EEPROM。这些设备通常需传输大量数据。大多数维护和控制应用程序，即I2C总线的常见用途，通常在100kHz频率（标准模式和快速模式）下运行。任何DW_apb_i2c设备均可连接至I2C总线，且每个设备均可与任意主设备通信，实现信息的双向传输。总线上必须至少有一个主设备（如微控制器或数字信号处理器），但亦可存在多个主设备，需通过仲裁机制确定总线所有权。多个主设备及仲裁机制将在本章后续部分详细阐述。I2C模块不支持SMBus及PMBus协议（分别用于系统管理与电源管理）。

DW_apb_i2c由AMBA APB从属接口、I2C接口及保持两者一致性的FIFO逻辑组成。组件各模块结构如图64所示。

图64. I2C块图



以下定义图64各模块的功能：

- **AMBA总线接口单元**— 将APB接口信号转换为通用接口，使寄存器文件与总线协议无关。

- **Register File** – Contains configuration registers and is the interface with software.
- **Slave State Machine** – Follows the protocol for a slave and monitors bus for address match.
- **Master State Machine** – Generates the I2C protocol for the master transfers.
- **Clock Generator** – Calculates the required timing to do the following:
 - Generate the **SCL** clock when configured as a master
 - Check for bus idle
 - Generate a START and a STOP
 - Setup the data and hold the data
- **Rx Shift** – Takes data into the design and extracts it in byte format.
- **Tx Shift** – Presents data supplied by CPU for transfer on the I2C bus.
- **Rx Filter** – Detects the events in the bus; for example, start, stop and arbitration lost.
- **Toggle** – Generates pulses on both sides and toggles to transfer signals across clock domains.
- **Synchronizer** – Transfers signals from one clock domain to another.
- **DMA Interface** – Generates the handshaking signals to the central DMA controller in order to automate the data transfer without CPU intervention.
- **Interrupt Controller** – Generates the raw interrupt and interrupt flags, allowing them to be set and cleared.
- **RX FIFO/TX FIFO** – Holds the RX FIFO and TX FIFO register banks and controllers, along with their status levels.

4.3.4. I2C Terminology

The following terms are used and are defined as follows:

4.3.4.1. I2C Bus Terms

The following terms relate to how the role of the I2C device and how it interacts with other I2C devices on the bus.

- **Transmitter** – the device that sends data to the bus. A transmitter can either be a device that initiates the data transmission to the bus (a master-transmitter) or responds to a request from the master to send data to the bus (a slave-transmitter).
- **Receiver** – the device that receives data from the bus. A receiver can either be a device that receives data on its own request (a master-receiver) or in response to a request from the master (a slave-receiver).
- **Master** – the component that initializes a transfer (START command), generates the clock **SCL** signal and terminates the transfer (STOP command). A master can be either a transmitter or a receiver.
- **Slave** – the device addressed by the master. A slave can be either receiver or transmitter.
- **Multi-master** – the ability for more than one master to co-exist on the bus at the same time without collision or data loss.
- **Arbitration** – the predefined procedure that authorizes only one master at a time to take control of the bus. For more information about this behaviour, refer to [Section 4.3.8](#).
- **Synchronization** – the predefined procedure that synchronizes the clock signals provided by two or more masters. For more information about this feature, refer to [Section 4.3.9](#).
- **SDA** – data signal line (Serial Data)
- **SCL** – clock signal line (Serial Clock)

- **寄存器文件**—含配置寄存器，是与软件的接口。
- **从状态机**—遵循从机协议，监控总线地址匹配。
- **主状态机**—生成主机传输所需的I2C协议。
- **时钟发生器**—计算以下操作所需的时序：
 - 配置为主设备时生成 **SCL**时钟
 - 检查总线空闲状态
 - 生成START和STOP信号
 - 设置并保持数据
- **Rx Shift** — 将数据导入设计并以字节格式提取。
- **Tx Shift** — 呈现CPU提供的数据以便在I2C总线上传输。
- **Rx Filter** — 检测总线上的事件；例如起始信号、停止信号及仲裁丢失。
- **Toggle** — 在两端产生脉冲并切换，用以跨时钟域传输信号。
- **Synchronizer** — 将信号从一个时钟域传输至另一时钟域。
- **DMA Interface**—向中央DMA控制器生成握手信号，实现数据传输自动化，无需CPU介入。
- **Interrupt Controller** — 生成原始中断信号与中断标志，允许设置及清除。
- **RX FIFO/TX FIFO** — 保持RX FIFO和TX FIFO寄存器组及其控制器，含状态级别。

4.3.4. I2C 术语

以下术语定义如下：

4.3.4.1. I2C总线术语

以下术语涉及I2C设备的角色及其与总线上的其他I2C设备的交互方式。

- **发送器**—向总线发送数据的设备。发送器可以是主动发起数据传输的设备（主发送器），也可以是响应主设备请求向总线发送数据的设备（从发送器）。
- 接收器—从总线接收数据的设备。接收器可以是自行请求接收数据的设备（主接收器），也可以是响应主设备请求的设备（从接收器）。
- 主设备—初始化传输（START命令）、生成时钟信号**SCL**并终止传输（STOP命令）的组件。主设备可以是发送器或接收器。
- 从设备—由主设备寻址的设备。从设备可以是接收器或发送器。
- **多主设备**—指总线上可同时存在多个主设备且不会发生冲突或数据丢失的能力。
- **仲裁**—指预定义程序，授权每次仅允许一个主设备控制总线。有关此行为的更多信息，参见第4.3.8节。
- **同步**—指预定义程序，用于同步两个或多个主设备所提供的时钟信号。
有关此功能的更多信息，参见第4.3.9节。
- **SDA** — 数据线（串行数据）
- **SCL** — 时钟线（串行时钟）

4.3.4.2. Bus Transfer Terms

The following terms are specific to data transfers that occur to/from the I2C bus.

- **START (RESTART)** – data transfer begins with a START or RESTART condition. The level of the **SDA** data line changes from high to low, while the **SCL** clock line remains high. When this occurs, the bus becomes busy.

i NOTE

START and RESTART conditions are functionally identical.

- **STOP** – data transfer is terminated by a STOP condition. This occurs when the level on the **SDA** data line passes from the low state to the high state, while the **SCL** clock line remains high. When the data transfer has been terminated, the bus is free or idle once again. The bus stays busy if a RESTART is generated instead of a STOP condition.

4.3.5. I2C Behaviour

The DW_apb_i2c can be controlled via software to be either:

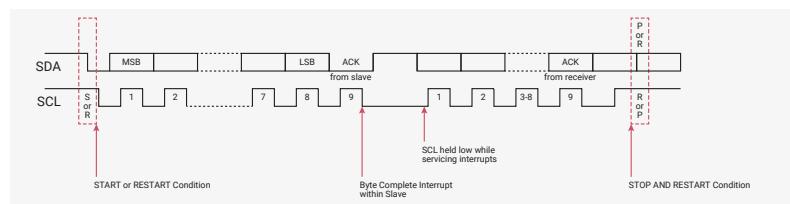
- An I2C master only, communicating with other I2C slaves; OR
- An I2C slave only, communicating with one or more I2C masters.

The master is responsible for generating the clock and controlling the transfer of data. The slave is responsible for either transmitting or receiving data to/from the master. The acknowledgement of data is sent by the device that is receiving data, which can be either a master or a slave. As mentioned previously, the I2C protocol also allows multiple masters to reside on the I2C bus and uses an arbitration procedure to determine bus ownership.

Each slave has a unique address that is determined by the system designer. When a master wants to communicate with a slave, the master transmits a START/RESTART condition that is then followed by the slave's address and a control bit (R/W) to determine if the master wants to transmit data or receive data from the slave. The slave then sends an acknowledge (ACK) pulse after the address.

If the master (master-transmitter) is writing to the slave (slave-receiver), the receiver gets one byte of data. This transaction continues until the master terminates the transmission with a STOP condition. If the master is reading from a slave (master-receiver), the slave transmits (slave-transmitter) a byte of data to the master, and the master then acknowledges the transaction with the ACK pulse. This transaction continues until the master terminates the transmission by not acknowledging (NACK) the transaction after the last byte is received, and then the master issues a STOP condition or addresses another slave after issuing a RESTART condition. This behaviour is illustrated in [Figure 65](#).

Figure 65. Data transfer on the I2C Bus



The DW_apb_i2c is a synchronous serial interface. The **SDA** line is a bidirectional signal and changes only while the **SCL** line is low, except for STOP, START, and RESTART conditions. The output drivers are open-drain or open-collector to perform wire-AND functions on the bus. The maximum number of devices on the bus is limited by only the maximum capacitance specification of 400 pF. Data is transmitted in byte packages.

The I2C protocols implemented in DW_apb_i2c are described in more details in [Section 4.3.6](#).

4.3.5.1. START and STOP Generation

When operating as an I2C master, putting data into the transmit FIFO causes the DW_apb_i2c to generate a START condition on the I2C bus. Writing a 1 to **IC_DATA_CMD.STOP** causes the DW_apb_i2c to generate a STOP condition on

4.3.4.2. 总线传输术语

以下术语专指发生于I2C总线上的数据传输。

- 启动（重新启动） – 数据传输以启动或重新启动条件开始。SDA数据线电平由高变为低，同时SCL时钟线保持高电平。当此情况发生时，总线处于忙碌状态。

i 注意

START 和 RESTART 条件在功能上完全相同。

- STOP – 通过 STOP 条件终止数据传输。当 SDA 数据线电平由低变高，而 SCL 时钟线保持高电平时，即发生此现象。数据传输终止后，总线重新处于空闲状态。若生成 RESTART 而非 STOP 条件，总线将保持忙碌状态。

4.3.5. I2C 行为

DW_apb_i2c 可通过软件控制，设定为以下任一模式：

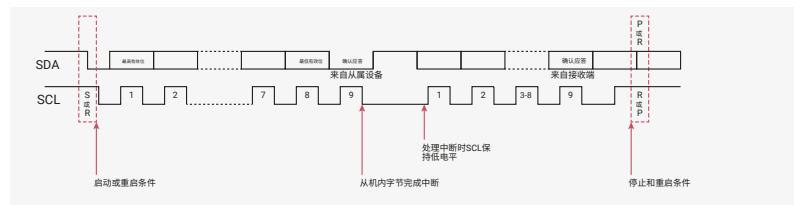
- 仅作为 I2C 主设备，与其他 I2C 从设备通信；或
- 仅作为 I2C 从设备，与一个或多个 I2C 主设备通信。

主设备负责产生时钟并控制数据传输。从设备负责向主设备发送或接收数据。数据确认由接收设备发出，该设备可为主设备或从设备。如前所述，I2C 协议亦允许多个主设备共存于 I2C 总线上，并通过仲裁程序来确定总线所有权。

每个从设备均拥有由系统设计者确定的唯一地址。当主设备欲与从设备通信时，主设备首先发送 START/RESTART 条件，随后发送从设备地址及控制位 (R/W)，以确定主设备是欲发送数据至从设备，还是从从设备接收数据。从设备接收地址后，会发送应答 (ACK) 信号。

若主设备（主发送方）向从设备（从接收方）写入数据，接收方将接收一个字节的数据。此传输过程持续进行，直到主设备通过发送 STOP 条件终止传输。若主设备从从设备读取数据（主接收方），从设备（从发送方）向主设备发送一个字节数据，主设备随后通过 ACK 信号确认该传输。此传输过程持续进行，直至主设备在接收最后一个字节后不再发送 ACK 信号（即发送 NACK），随后主设备发送 STOP 条件终止传输，或在发送 RESTART 条件后寻址另一从设备。该行为如图 65 所示。

图 65. I2C 总线上的数据传输



DW_apb_i2c 为同步串行接口。SDA 线为双向信号，仅在 SCL 线为低电平时发生变化，停止 (STOP)、起始 (START) 及重新起始 (RESTART) 条件除外。输出驱动为开漏或开集电极，以在总线上实现线与 (wire-AND) 功能。总线上设备的最大数量仅受最大电容规格 400 pF 的限制。数据以字节包形式传输。

DW_apb_i2c 中实现的 I2C 协议详见第 4.3.6 节。

4.3.5.1. 起始 (START) 与停止 (STOP) 信号的产生

作为 I2C 主设备时，向传输 FIFO 写入数据会触发 DW_apb_i2c 在 I2C 总线上生成起始 (START) 信号。向 IC_DATA_CMD.ST_OP 写入 1 会使 DW_apb_i2c 在

the I²C bus; a STOP condition is not issued if this bit is not set, even if the transmit FIFO is empty.

When operating as a slave, the DW_apb_i2c does not generate START and STOP conditions, as per the protocol. However, if a read request is made to the DW_apb_i2c, it holds the SCL line low until read data has been supplied to it. This stalls the I²C bus until read data is provided to the slave DW_apb_i2c, or the DW_apb_i2c slave is disabled by writing a 0 to IC_ENABLE.ENABLE.

4.3.5.2. Combined Formats

The DW_apb_i2c supports mixed read and write combined format transactions in both 7-bit and 10-bit addressing modes. The DW_apb_i2c does not support mixed address and mixed address format—that is, a 7-bit address transaction followed by a 10-bit address transaction or vice versa—combined format transactions. To initiate combined format transfers, IC_CON.IC_RESTART_EN should be set to 1. With this value set and operating as a master, when the DW_apb_i2c completes an I²C transfer, it checks the transmit FIFO and executes the next transfer. If the direction of this transfer differs from the previous transfer, the combined format is used to issue the transfer. If the transmit FIFO is empty when the current I²C transfer completes:

- IC_DATA_CMD.STOP is checked and:
 - If set to 1, a STOP bit is issued.
 - If set to 0, the SCL is held low until the next command is written to the transmit FIFO.

For more details, refer to [Section 4.3.7](#).

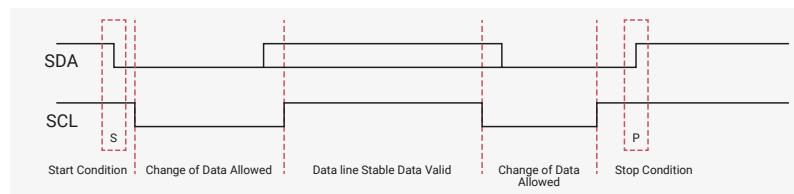
4.3.6. I²C Protocols

The DW_apb_i2c has the protocols discussed in this section.

4.3.6.1. START and STOP Conditions

When the bus is idle, both the SCL and SDA signals are pulled high through external pull-up resistors on the bus. When the master wants to start a transmission on the bus, the master issues a START condition. This is defined to be a high-to-low transition of the SDA signal while SCL is 1. When the master wants to terminate the transmission, the master issues a STOP condition. This is defined to be a low-to-high transition of the SDA line while SCL is 1. [Figure 66](#) shows the timing of the START and STOP conditions. When data is being transmitted on the bus, the SDA line must be stable when SCL is 1.

Figure 66. I²C START and STOP Condition



NOTE

The signal transitions for the START/STOP conditions, as depicted in [Figure 66](#), reflect those observed at the output signals of the Master driving the I²C bus. Care should be taken when observing the SDA/SCL signals at the input signals of the Slave(s), because unequal line delays may result in an incorrect SDA/SCL timing relationship.

4.3.6.2. Addressing Slave Protocol

There are two address formats: the 7-bit address format and the 10-bit address format.

I²C总线上生成停止（STOP）信号；若未设置此位，即使传输FIFO为空，也不会发出停止（STOP）信号。

作为从设备运行时，DW_apb_i2c 不会产生协议中的 START 和 STOP 条件。

但如果向 DW_apb_i2c 发出读请求，其会将 SCL 线保持低电平，直到向其提供读取数据。

这将使 I²C 总线阻塞，直到向从设备 DW_apb_i2c 提供读取数据，或通过向 IC_ENABLE.ENABLE 寄存器写入 0 禁用 DW_apb_i2c 从设备。

4.3.5.2. 组合格式

DW_apb_i2c 支持 7 位和 10 位寻址模式下的混合读写组合格式事务。DW_apb_i2c 不支持混合地址及混合地址格式的组合格式事务，即先进行 7 位地址事务后再进行 10 位地址事务，或反之。要启动组合格式传输，应将 IC_CON.IC_RESTART_EN 设置为 1。当该值设置且作为主设备运行时，DW_apb_i2c 在完成一次I²C传输后，会检查发送FIFO并执行下一次传输。如果此次传输方向与上一次传输不同，则采用组合格式发起该传输。当当前I²C传输完成且发送FIFO为空时：

- 将检查IC_DATA_CMD.STOP位，且：
 - 若该位设为1，则发送STOP位。
 - 若该位设为0，则 SCL 线保持低电平，直到下一条命令写入发送FIFO。

详细信息请参见第4.3.7节。

4.3.6. I²C 协议

DW_apb_i2c支持本节所述的协议。

4.3.6.1. START与STOP条件

当总线处于空闲状态时，SCL和SDA信号线通过外部上拉电阻被拉高。当主设备欲在总线上开始传输时，主设备发出START条件。定义为在 SCL 为 1 时，SDA 信号由高电平到低电平的跳变。当主设备欲终止传输时，主设备应发出STOP条件。定义为在 SCL 为 1 时，SDA 线路由低电平跳变至高电平。图66显示了START和STOP条件的时序。当数据在总线上传输时，SCL 为 1 期间，SDA 线路必须保持稳定。

图66。I²C START 和 STOP 条件



① 注意

图66所示的START/STOP条件信号跳变，反映了主设备驱动I²C总线时的输出信号。观察从设备输入端的 SDA/SCL 信号时须谨慎，因线路延迟不一致可能导致 SDA/SCL 时序关系错误。

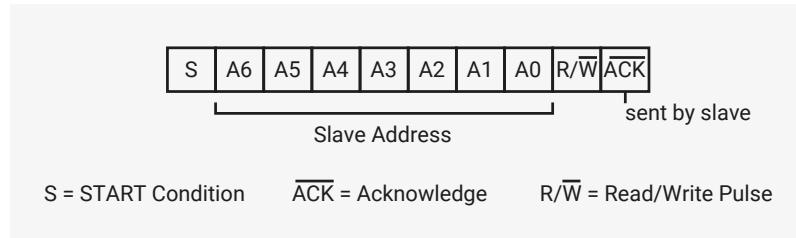
4.3.6.2. 从设备协议寻址

地址格式分为两种：7位地址格式和10位地址格式。

4.3.6.2.1. 7-bit Address Format

During the 7-bit address format, the first seven bits (bits 7:1) of the first byte set the slave address and the LSB bit (bit 0) is the R/W bit as shown in Figure 67. When bit 0 (R/W) is set to 0, the master writes to the slave. When bit 0 (R/W) is set to 1, the master reads from the slave.

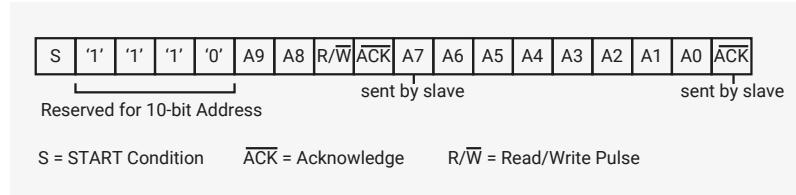
Figure 67. I_C 7-bit Address Format



4.3.6.2.2. 10-bit Address Format

During 10-bit addressing, two bytes are transferred to set the 10-bit address. The transfer of the first byte contains the following bit definition. The first five bits (bits 7:3) notify the slaves that this is a 10-bit transfer followed by the next two bits (bits 2:1), which set the slaves address bits 9:8, and the LSB bit (bit 0) is the R/W bit. The second byte transferred sets bits 7:0 of the slave address. Figure 68 shows the 10-bit address format.

Figure 68. 10-bit Address Format



This table defines the special purpose and reserved first byte addresses.

**Table 448. I₂C/SMBus
Definition of Bits in
First Byte**

Slave Address	R/W Bit	Description
0000 000	0	General Call Address. DW_apb_i2c places the data in the receive buffer and issues a General Call interrupt.
0000 000	1	START byte. For more details, refer to Section 4.3.6.4 .
0000 001	X	CBUS address. DW_apb_i2c ignores these accesses.
0000 010	X	Reserved.
0000 011	X	Reserved.
0000 1XX	X	High-speed master code (for more information, refer to Section 4.3.8).
1111 1XX	X	Reserved.
1111 0XX	X	10-bit slave addressing.
0001 000	X	SMbus Host (not supported)
0001 100	X	SMBus Alert Response Address (not supported)
1100 001	X	SMBus Device Default Address (not supported)

DW_apb_i2c does not restrict you from using these reserved addresses. However, if you use these reserved addresses,

4.3.6.2.1. 7位地址格式

在7位地址格式中，首字节的高七位（第7至第1位）表示从设备地址，最低位（第0位）为读/写标志位，如图67所示。当第0位（读/写）为0时，主设备向从设备写入数据。当第0位（读/写）为1时，主设备从从设备读取数据。

图67. I_C 7位地址格式



4.3.6.2.2. 10位地址格式

在10位寻址过程中，传输两个字节以设置10位地址。第一个字节的传输内容包括以下位定义。前五位（位7至位3）通知从设备此次操作作为10位寻址，随后两位（位2至位1）设置从设备地址的第9位和第8位，最低有效位（位0）为读写位。第二个传输字节设置从设备地址的位7至位0。图68展示了10位地址格式。

图68. 10位地址格式



本表定义了特殊用途及保留的首字节地址。

表448。I_C/SMBus首字节各位定义

从设备地址	读写位	描述
0000 000	0	通用呼叫地址。DW_apb_i2c将数据置于接收缓冲区并触发通用调用中断。
0000 000	1	START 字节。更多详情，请参见第4.3.6.4节。
0000 001	X	CBUS 地址。DW_apb_i2c 忽略此类访问。
0000 010	X	保留。
0000 011	X	保留。
0000 1XX	X	高速主机代码（详见第4.3.8节）。
1111 1XX	X	保留。
1111 0XX	X	10位从机地址。
0001 000	X	SMBus 主机（不支持）
0001 100	X	SMBus 警报响应地址（不支持）
1100 001	X	SMBus 设备默认地址（不支持）

DW_apb_i2c 不限制您使用这些保留地址。但若您使用这些保留地址，

you may run into incompatibilities with other I²C components.

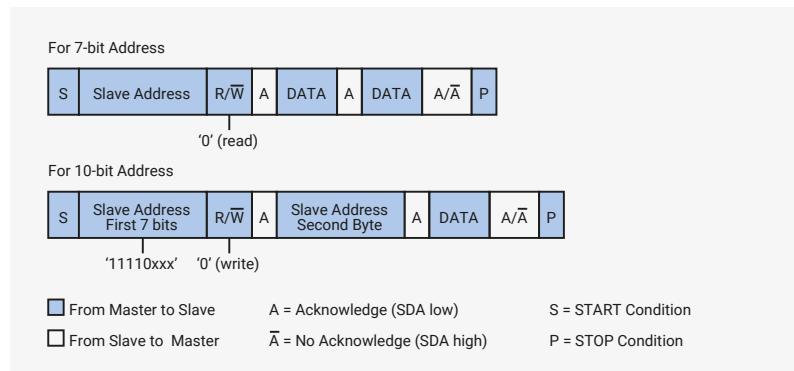
4.3.6.3. Transmitting and Receiving Protocol

The master can initiate data transmission and reception to/from the bus, acting as either a master-transmitter or master-receiver. A slave responds to requests from the master to either transmit data or receive data to/from the bus, acting as either a slave-transmitter or slave-receiver, respectively.

4.3.6.3.1. Master-Transmitter and Slave-Receiver

All data is transmitted in byte format, with no limit on the number of bytes transferred per data transfer. After the master sends the address and R/W bit or the master transmits a byte of data to the slave, the slave-receiver must respond with the acknowledge signal (ACK). When a slave-receiver does not respond with an ACK pulse, the master aborts the transfer by issuing a STOP condition. The slave must leave the SDA line high so that the master can abort the transfer. If the master-transmitter is transmitting data as shown in [Figure 69](#), then the slave-receiver responds to the master-transmitter with an acknowledge pulse after every byte of data is received.

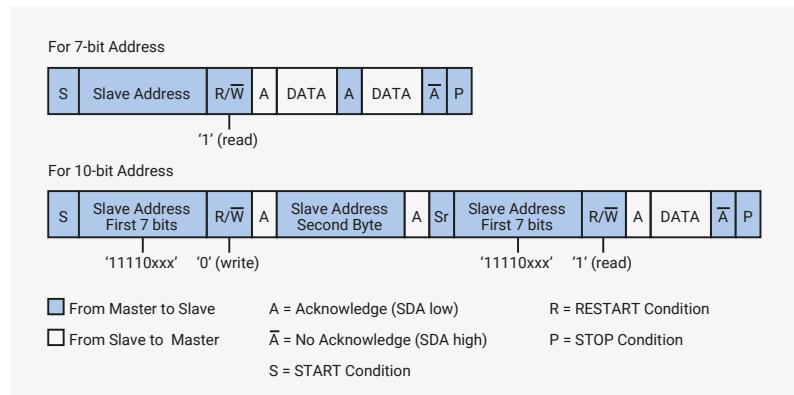
Figure 69. I²C Master-Transmitter Protocol



4.3.6.3.2. Master-Receiver and Slave-Transmitter

If the master is receiving data as shown in [Figure 70](#), then the master responds to the slave-transmitter with an acknowledge pulse after a byte of data has been received, except for the last byte. This is the way the master-receiver notifies the slave-transmitter that this is the last byte. The slave-transmitter relinquishes the SDA line after detecting the No Acknowledge (NACK) so that the master can issue a STOP condition.

Figure 70. I²C Master-Receiver Protocol



When a master does not want to relinquish the bus with a STOP condition, the master can issue a RESTART condition. This is identical to a START condition except it occurs after the ACK pulse. Operating in master mode, the DW_apb_i2c can then communicate with the same slave using a transfer of a different direction. For a description of the combined format transactions that the DW_apb_i2c supports, refer to [Section 4.3.5.2](#).

可能导致与其他 I2C 组件的不兼容。

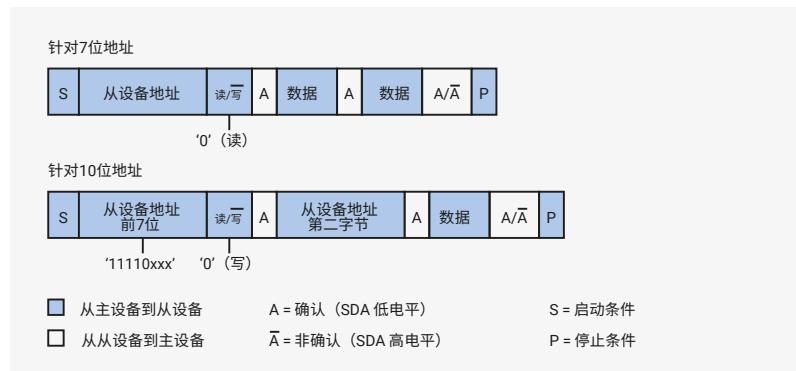
4.3.6.3 传输与接收协议

主机可发起总线上的数据传输与接收，既可作为主发送器，也可作为主接收器。从设备响应主设备的请求，执行数据的发送或接收操作，分别作为从设备发送方或从设备接收方。

4.3.6.3.1. 主设备发送方与从设备接收方

所有数据均以字节格式传输，单次数据传输的字节数无限制。主设备发送地址和读写位后，或主设备向从设备传输一个字节数据后，从设备接收方必须发送应答信号（ACK）。当从设备接收方未响应ACK信号时，主设备通过发送停止条件终止传输。从设备必须保持 SDA 线高电平，以便主设备能够终止传输。如果主设备发送方如图69所示传输数据，则从设备接收方在每接收一个字节数据后向主设备发送应答信号。

图69。I2C主控发送协议



4.3.6.3.2. 主接收器与从传输器

如图70所示，若主设备正在接收数据，则在接收完每个数据字节后（除最后一个字节外），主设备会向从传输器发送确认脉冲。这是主接收器通知从发送器这是最后一个字节的方式。从发送器在检测到无应答（NACK）后释放 SDA线，以便主设备发出停止条件。

图70。I2C主设备-接收器协议



当主设备不希望通过停止条件释放总线时，可发出重启条件。

此条件与起始条件相同，但发生在应答脉冲之后。在主模式下运行时，DW_apb_i2c可通过方向不同的传输与同一从设备通信。有关DW_apb_i2c支持的复合格式事务的描述，请参见第4.3.5.2节。

NOTE

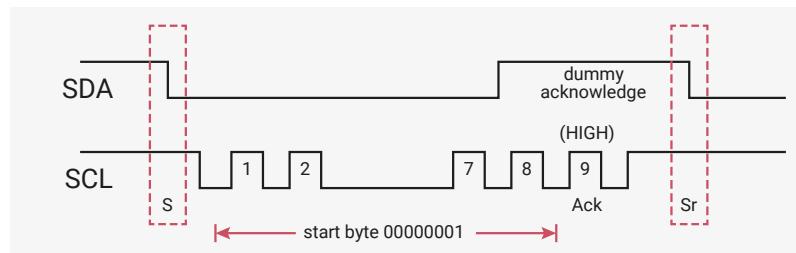
The DW_apb_i2c must be completely disabled before the target slave address register ([IC_TAR](#)) can be reprogrammed.

4.3.6.4. START BYTE Transfer Protocol

The START BYTE transfer protocol is set up for systems that do not have an on-board dedicated I2C hardware module. When the DW_apb_i2c is addressed as a slave, it always samples the I2C bus at the highest speed supported so that it never requires a START BYTE transfer. However, when DW_apb_i2c is a master, it supports the generation of START BYTE transfers at the beginning of every transfer in case a slave device requires it.

This protocol consists of seven zeros being transmitted followed by a one, as illustrated in [Figure 71](#). This allows the processor that is polling the bus to under-sample the address phase until zero is detected. Once the microcontroller detects a zero, it switches from the under sampling rate to the correct rate of the master.

Figure 71. I2C Start Byte Transfer



The START BYTE procedure is as follows:

1. Master generates a START condition.
2. Master transmits the START byte (0000 0001).
3. Master transmits the ACK clock pulse. (Present only to conform with the byte handling format used on the bus)
4. No slave sets the ACK signal to zero.
5. Master generates a RESTART (R) condition.

A hardware receiver does not respond to the START BYTE because it is a reserved address and resets after the RESTART condition is generated.

4.3.7. Tx FIFO Management and START, STOP and RESTART Generation

When operating as a master, the DW_apb_i2c component supports the mode of Tx FIFO management illustrated in [Figure 72](#)

4.3.7.1. Tx FIFO Management

The component does not generate a STOP if the Tx FIFO becomes empty; in this situation the component holds the [SCL](#) line low, stalling the bus until a new entry is available in the Tx FIFO. A STOP condition is generated only when the user specifically requests it by setting bit nine (Stop bit) of the command written to [IC_DATA_CMD](#) register. [Figure 72](#) shows the bits in the [IC_DATA_CMD](#) register.

注意

在重新编程目标从设备地址寄存器 (IC_TAR) 之前，必须完全禁用 DW_apb_i2c。

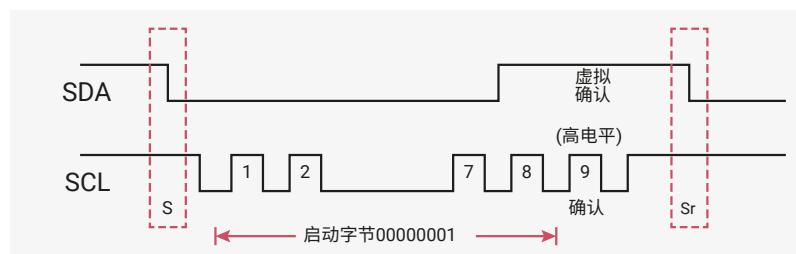
4.3.6.4. START BYTE 传输协议

START BYTE 传输协议适用于无板载专用 I2C 硬件模块的系统。

当 DW_apb_i2c 作为从设备被寻址时，始终以支持的最高速度采样 I2C 总线，因此不需要 START BYTE 传输。但当 DW_apb_i2c 作为主设备时，支持在每次传输开始时生成 START BYTE 传输，以应对从设备的需求。

该协议包括先发送七个零位，随后发送一个一位，如图 71 所示。这使得轮询总线的处理器可在地址阶段低速采样，直到检测到零位。检测到零位后，微控制器将采样速率从低速切换至主设备的正确速率。

图 71. I2C 启动字节传输



START BYTE 过程如下：

1. 主机生成START起始条件。
2. 主机发送START字节（0000 0001）。
3. 主机发送ACK应答时钟脉冲。（仅为符合总线上字节处理格式而存在）
4. 无任何从机将ACK信号置零。
5. 主机生成RESTART（重复启动）条件。

硬件接收器不会响应START字节，因为该地址被保留，并在生成RESTART条件后复位。

4.3.7. 发送 FIFO 管理及 START、STOP 和 RESTART 生成

作为主机操作时，DW_apb_i2c组件支持图72中所示的Tx FIFO管理模式。

4.3.7.1. Tx FIFO管理

当Tx FIFO变为空时，组件不会生成STOP条件；此时组件将SCL线保持低电平，暂停总线，直到Tx FIFO中有新数据。仅当用户通过设置写入IC_DATA_CMD寄存器命令的第九位（停止位）时，才会生成STOP条件。图72显示了IC_DATA_CMD寄存器中的各个位。

Figure 72.
IC_DATA_CMD
Register

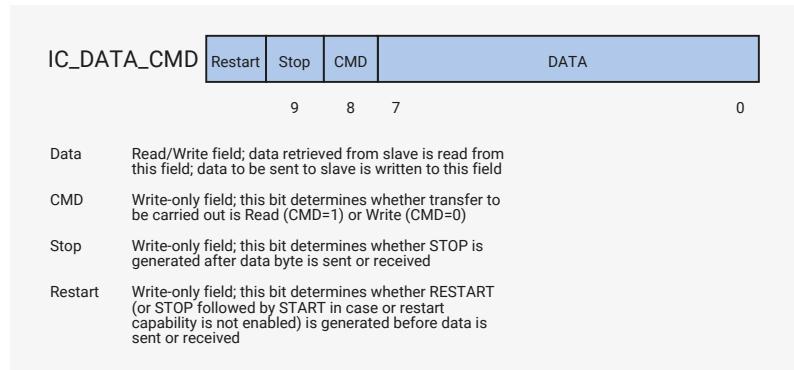


Figure 73 illustrates the behaviour of the DW_apb_i2c when the Tx FIFO becomes empty while operating as a master transmitter, as well as showing the generation of a STOP condition.

Figure 73. Master Transmitter - Tx FIFO Empties/STOP Generation

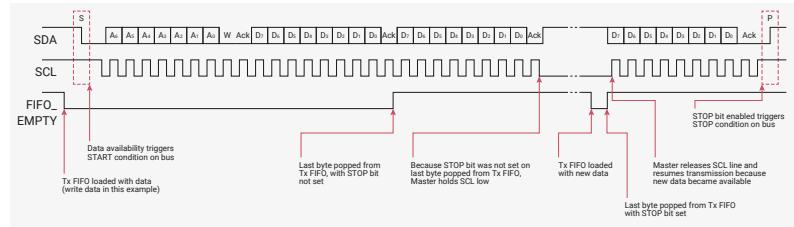


Figure 74 illustrates the behaviour of the DW_apb_i2c when the Tx FIFO becomes empty while operating as a master receiver, as well as showing the generation of a STOP condition.

Figure 74. Master Receiver - Tx FIFO Empties/STOP Generation

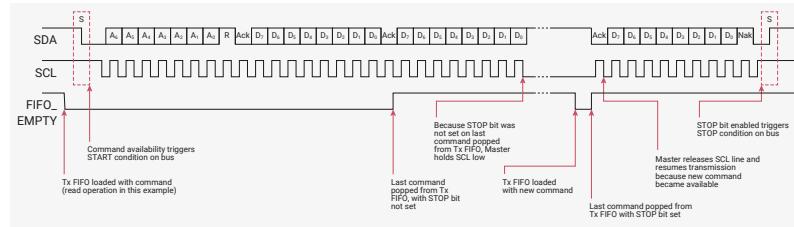


Figure 75 and Figure 76 illustrate configurations where the user can control the generation of RESTART conditions on the I2C bus. If bit 10 (Restart) of the IC_DATA_CMD register is set and the restart capability is enabled (IC_RESTART_EN=1), a RESTART is generated before the data byte is written to or read from the slave. If the restart capability is not enabled a STOP followed by a START is generated in place of the RESTART. Figure 75 illustrates this situation during operation as a master transmitter.

Figure 75. Master Transmitter – Restart Bit of IC_DATA_CMD Is Set

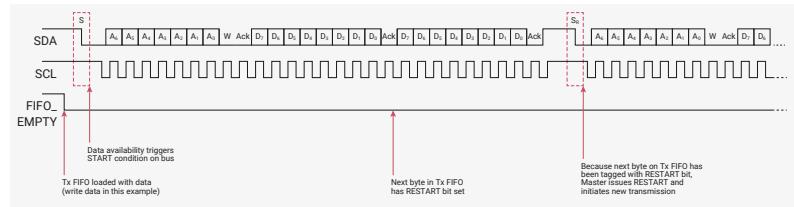


Figure 76 illustrates the same situation, but during operation as a master receiver.

Figure 76. Master Receiver – Restart Bit of IC_DATA_CMD Is Set

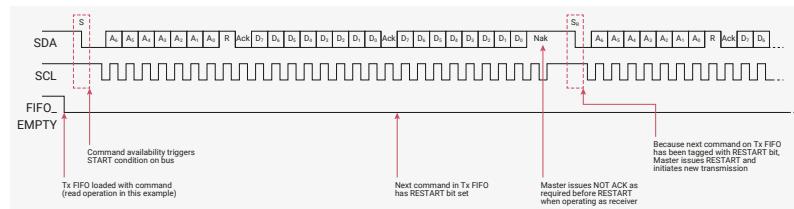


Figure 77 illustrates operation as a master transmitter where the Stop bit of the IC_DATA_CMD register is set and the Tx FIFO is not empty

图72。
IC_DATA_CMD
寄存器



图73说明了 DW_apb_i2c 作为主机发送器时，发送FIFO（Tx FIFO）为空时的行为，以及停止（STOP）条件的生成。

图73。主控
发送器 - Tx FIFO
清空/STOP
生成

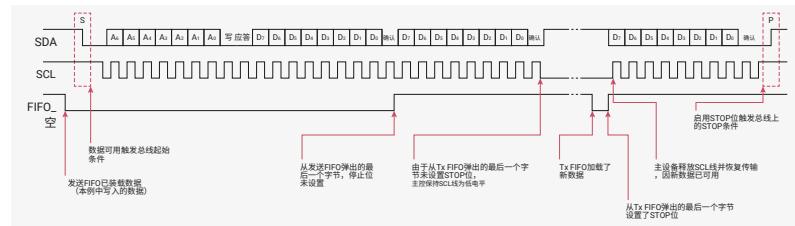


图74展示了DW_apb_i2c作为主控接收器时，Tx FIFO变为空的行为及STOP条件的生成。

图 74。主机
接收器 - 发送 FIFO
清空/停止
生成

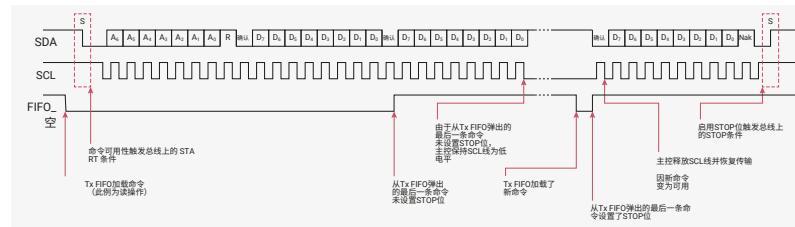


图 75 和图 76 展示了用户可控制 I2C 总线上 RESTART 条件生成的配置。如果寄存器 IC_DATA_CMD 中的第 10 位 (Restart) 被设置，且重启功能已启用 (IC_RESTART_EN=1)，则在数据字节写入或读取从机之前将生成 RESTART。如果未启用重启功能，则会先生成 STOP，随后生成 START 以代替 RESTART。图 75 展示了作为主机发送方在此情况下的操作。

图 75。主机
发送方 - 重启
IC_DATA_CMD 寄存器
的位被设置

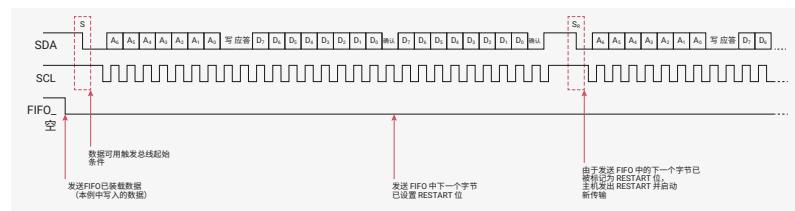


图76展示了作为主设备接收器时的相同情况。

图76。主接收
器--IC_DATA_CMD
寄存器的Restart
位已设置

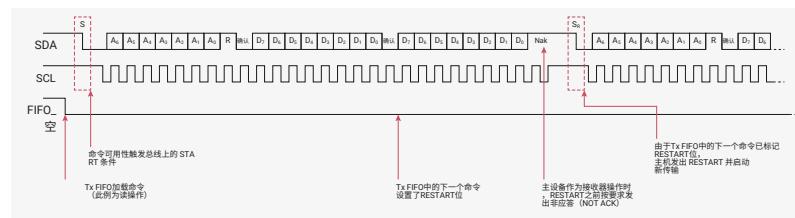


图77展示了作为主设备发送器时，IC_DATA_CMD寄存器Stop位设置且Tx FIFO非空的操作

Figure 77. Master Transmitter – Stop Bit of IC_DATA_CMD Set/Tx FIFO Not Empty

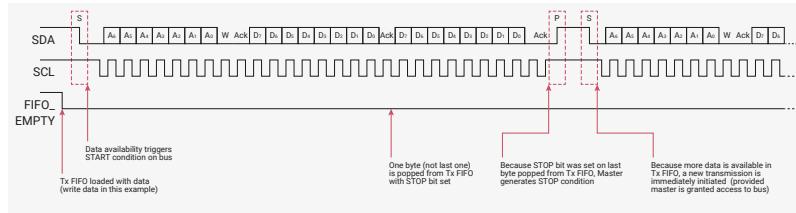


Figure 78 illustrates operation as a master transmitter where the first byte loaded into the Tx FIFO is allowed to go empty with the Restart bit set

Figure 78. Master Transmitter – First Byte Loaded Into Tx FIFO Allowed to Empty, Restart Bit Set

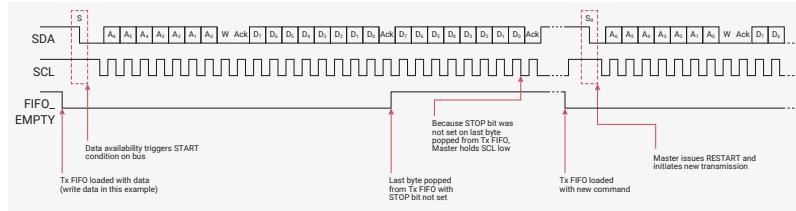


Figure 79 illustrates operation as a master receiver where the Stop bit of the IC_DATA_CMD register is set and the Tx FIFO is not empty

Figure 79. Master Receiver – Stop Bit of IC_DATA_CMD Set/Tx FIFO Not Empty

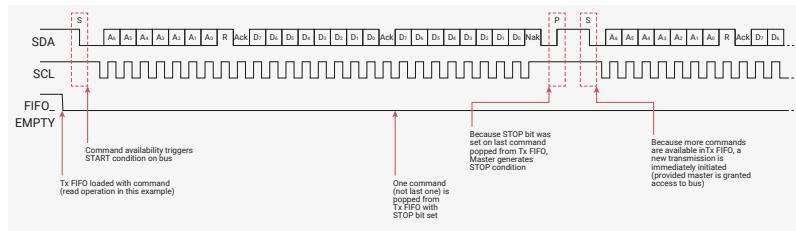
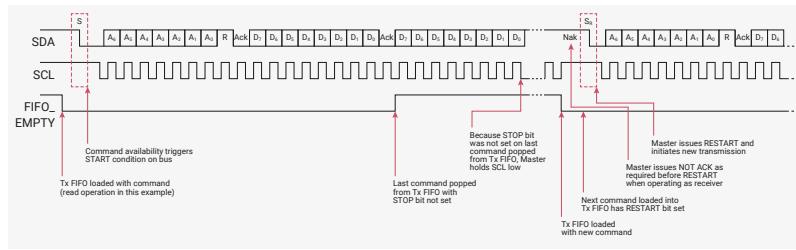


Figure 80 illustrates operation as a master receiver where the first command loaded after the Tx FIFO is allowed to empty and the Restart bit is set

Figure 80. Master Receiver – First Command Loaded After Tx FIFO Allowed to Empty/Restart Bit Set



4.3.8. Multiple Master Arbitration

The DW_apb_i2c bus protocol allows multiple masters to reside on the same bus. If there are two masters on the same I2C-bus, there is an arbitration procedure if both try to take control of the bus at the same time by generating a START condition at the same time. Once a master (for example, a microcontroller) has control of the bus, no other master can take control until the first master sends a STOP condition and places the bus in an idle state.

Arbitration takes place on the **SDA** line, while the **SCL** line is one. The master, which transmits a one while the other master transmits zero, loses arbitration and turns off its data output stage. The master that lost arbitration can continue to generate clocks until the end of the byte transfer. If both masters are addressing the same slave device, the arbitration could go into the data phase.

Upon detecting that it has lost arbitration to another master, the DW_apb_i2c will stop generating **SCL** (will disable the output driver). **Figure 81** illustrates the timing of when two masters are arbitrating on the bus.

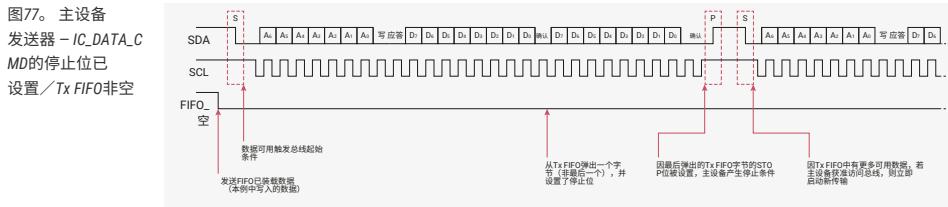


图77展示了主设备发送器 - IC_DATA_C 的时序图

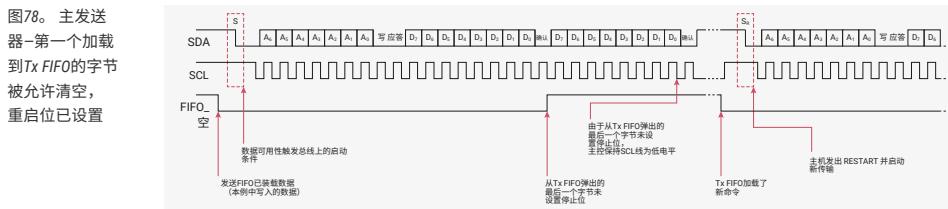


图78展示了主设备发送器的操作，其中IC_DATA_CMD寄存器的停止位已设置且Tx FIFO非空

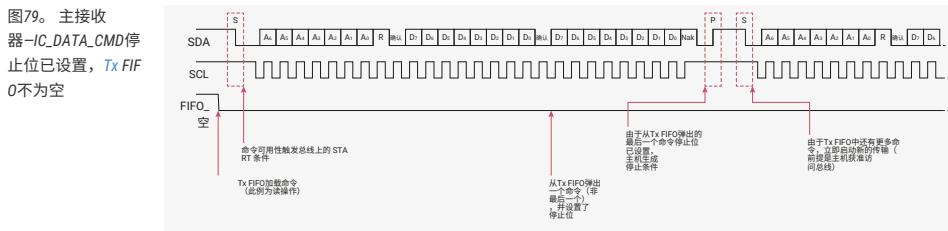
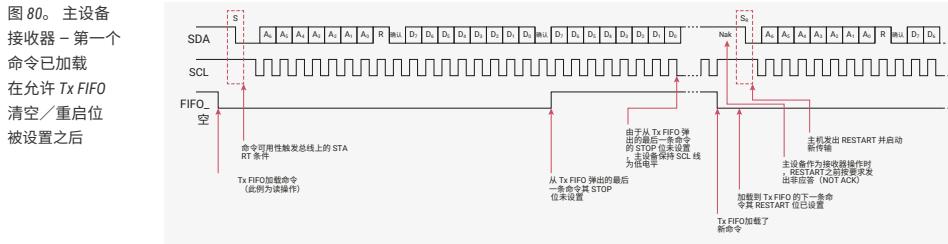


图79展示了主接收器的操作，其中IC_DATA_CMD寄存器的停止位已设置且Tx FIFO不为空



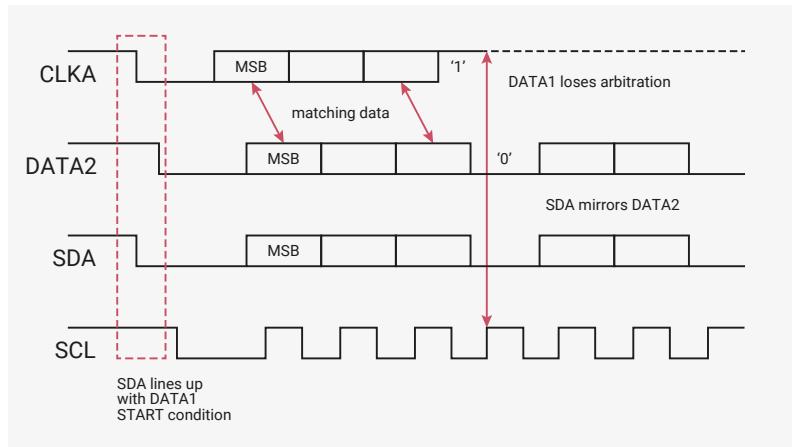
4.3.8. 多主机仲裁

DW_apb_i2c 总线协议允许多个主设备共存于同一总线上。当同一 I2C 总线上存在两个主设备且它们同时尝试生成 START 条件以控制总线时，将启动仲裁流程。一旦某一主设备（例如微控制器）获得总线控制权，其他主设备在该主设备发送 STOP 条件并使总线处于空闲状态之前，均无法获得控制权。

仲裁过程发生在 SDA 线上，而 SCL 线保持为高电平。发送“1”信号的主设备在另一主设备发送“0”时失去仲裁权，并关闭其数据输出阶段。失去仲裁的主设备可以继续生成时钟，直至字节传输结束。若两个主设备均寻址同一从设备，仲裁可能进入数据阶段。

检测到被另一主设备仲裁失败后，DW_apb_i2c 将停止生成 SCL（禁用输出驱动）。图81展示了两个主设备在总线上进行仲裁的时序。

Figure 81. Multiple Master Arbitration



Control of the bus is determined by address or master code and data sent by competing masters, so there is no central master nor any order of priority on the bus.

Arbitration is not allowed between the following conditions:

- A RESTART condition and a data bit
- A STOP condition and a data bit
- A RESTART condition and a STOP condition

NOTE

Slaves are not involved in the arbitration process.

4.3.9. Clock Synchronization

When two or more masters try to transfer information on the bus at the same time, they must arbitrate and synchronize the **SCL** clock. All masters generate their own clock to transfer messages. Data is valid only during the high period of **SCL** clock. Clock synchronization is performed using the wired-AND connection to the **SCL** signal. When the master transitions the **SCL** clock to zero, the master starts counting the low time of the **SCL** clock and transitions the **SCL** clock signal to one at the beginning of the next clock period. However, if another master is holding the **SCL** line to 0, then the master goes into a HIGH wait state until the **SCL** clock line transitions to one.

All masters then count off their high time, and the master with the shortest high time transitions the **SCL** line to zero. The masters then count out their low time and the one with the longest low time forces the other masters into a HIGH wait state. Therefore, a synchronized **SCL** clock is generated, which is illustrated in Figure 82. Optionally, slaves may hold the **SCL** line low to slow down the timing on the I²C bus.

Figure 82. Multi-Master Clock Synchronization

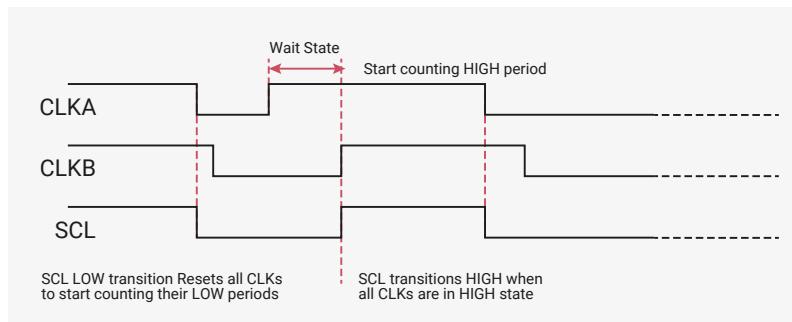
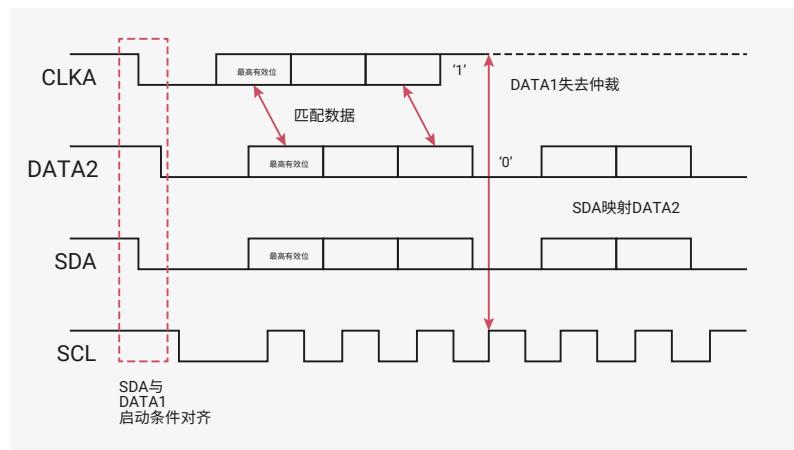


图81。多主设备
仲裁



总线控制由竞争主设备发送的地址或主代码及数据决定，因此总线上不存在中央主设备或优先级顺序。

如下情况不允许发生仲裁：

- 重新启动（RESTART）条件与数据位
- 停止（STOP）条件与数据位
- 重新启动（RESTART）条件与停止（STOP）条件

① 注意

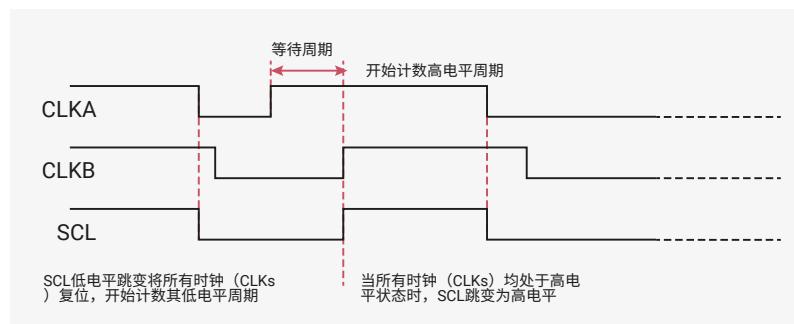
从设备不参与仲裁过程。

4.3.9. 时钟同步

当两个或多个主设备同时尝试在总线上传输信息时，必须对 SCL时钟进行仲裁和同步。所有主设备均生成各自的时钟以传输消息。数据仅在 SCL时钟的高电平期间有效。时钟同步通过对 SCL信号的有线与（wired-AND）连接实现。当主设备将 SCL时钟拉低至0时，开始计时 SCL时钟的低电平时间，并在下一时钟周期开始时将 SCL时钟信号置高。但如果另一主设备持续将 SCL线路保持为0，则该主设备将进入高电平等待状态，直至 SCL时钟线路变为1。

所有主设备随后计数其高电平时间，最短高电平时间的主设备将 SCL线路拉低。然后主设备计数其低电平时间，最长低电平时间的主设备强制其它主设备进入高电平等待状态。因此生成了一个同步的 SCL时钟，如图82所示。从设备可选择将SCL线拉低，以减缓I2C总线上的时序。

图82。多主时
钟同步



4.3.10. Operation Modes

This section provides information on operation modes.

i NOTE

It is important to note that the DW_apb_i2c should only be set to operate as an I2C Master, or I2C Slave, but not both simultaneously. This is achieved by ensuring that `IC_CON.IC_SLAVE_DISABLE` and `IC_CON.MASTER_MODE` are never set to zero and one, respectively.

4.3.10.1. Slave Mode Operation

This section discusses slave mode procedures.

4.3.10.1.1. Initial Configuration

To use the DW_apb_i2c as a slave, perform the following steps:

1. Disable the DW_apb_i2c by writing a '0' to `IC_ENABLE.ENABLE`.
2. Write to the `IC_SAR` register (bits 9:0) to set the slave address. This is the address to which the DW_apb_i2c responds.
3. Write to the `IC_CON` register to specify which type of addressing is supported (7-bit or 10-bit by setting bit 3). Enable the DW_apb_i2c in slave-only mode by writing a '0' into bit six (`IC_SLAVE_DISABLE`) and a '0' to bit zero (`MASTER_MODE`).

i NOTE

Slaves and masters do not have to be programmed with the same type of addressing 7-bit or 10-bit address. For instance, a slave can be programmed with 7-bit addressing and a master with 10-bit addressing, and vice versa.

1. Enable the DW_apb_i2c by writing a '1' to `IC_ENABLE.ENABLE`.

i NOTE

Depending on the reset values chosen, steps two and three may not be necessary because the reset values can be configured. For instance, if the device is only going to be a master, there would be no need to set the slave address because you can configure DW_apb_i2c to have the slave disabled after reset and to enable the master after reset. The values stored are static and do not need to be reprogrammed if the DW_apb_i2c is disabled.

– WARNING

It is recommended that the DW_apb_i2c Slave be brought out of reset only when the I2C bus is IDLE. De-asserting the reset when a transfer is ongoing on the bus causes internal synchronization flip-flops used to synchronize `SDA` and `SCL` to toggle from a reset value of one to the actual value on the bus. This can result in `SDA` toggling from one to zero while `SCL` is one, thereby causing a false START condition to be detected by the DW_apb_i2c Slave. This scenario can also be avoided by configuring the DW_apb_i2c with `IC_SLAVE_DISABLE` = 1 and `MASTER_MODE` = 1 so that the Slave interface is disabled after reset. It can then be enabled by programming `IC_CON[0]` = 0 and `IC_CON[6]` = 0 after the internal `SDA` and `SCL` have synchronized to the value on the bus; this takes approximately six `ic_clk` cycles after reset de-assertion.

4.3.10. 操作模式

本节提供有关操作模式的信息。

i 注意

务必注意，DW_apb_i2c应仅配置为I2C主机或I2C从机，不得同时兼备两者。通过确保IC_CON.IC_SLAVE_DISABLE不设置为零且IC_CON.MASTER_MODE不设置为一，达到此目的。

4.3.10.1 从机模式操作

本节讨论从机模式的操作程序。

4.3.10.1.1 初始配置

要将DW_apb_i2c用作从机，请执行以下步骤：

1. 通过向 IC_ENABLE.ENABLE 寄存器写入‘0’来禁用 DW_apb_i2c。
 2. 向 IC_SAR 寄存器（位 9:0）写入以设置从机地址。该地址为 DW_apb_i2c 响应的地址。
 3. 向 IC_CON 寄存器写入以指定支持的寻址类型（通过设置第3位选择7位或10位寻址）。
- 通过向第六位（IC_SLAVE_DISABLE）写入‘0’，以及向第零位（MASTER_MODE）写入‘0’，启用 DW_apb_i2c 的仅从机模式。

i 注意

从机和主机可以使用不同的寻址类型，支持7位或10位地址。例如，主机可设置为10位寻址，从机可设置为7位寻址，反之亦然。

1. 通过向 IC_ENABLE.ENABLE 寄存器写入‘1’来启用 DW_apb_i2c。

i 注意

根据复位时选定的初始值，步骤二和三可能不必执行，因为复位值可被配置。例如，如果设备仅用作主设备，则无需设置从设备地址，因为可以配置 DW_apb_i2c 在复位后禁用从设备，并在复位后启用主设备。

存储的值为静态值，若 DW_apb_i2c 被禁用，则无需重新编程。

— 警告

建议仅在 I2C 总线处于空闲（IDLE）状态时，才将 DW_apb_i2c 从复位状态释放。若在传输过程中取消复位，内部用于同步 SDA 和 SCL 的同步触发器将从复位值“1”切换为总线上的实际值，从而可能导致在 SCL 为“1”时，SDA 从“1”切换为“0”，使 DW_apb_i2c 从设备误判为虚假起始条件。该情况亦可通过配置 DW_apb_i2c，将 IC_SLAVE_DISABLE 设为 1 且 MASTER_MODE 设为 1，在复位后禁用从设备接口来避免。随后，通过编程设置 IC_CON[0] = 0 和 IC_CON[6] = 0 启用该功能，前提是内部 SDA 和 SCL 已与总线上的信号同步；此过程大约需要六个

ic_clk 周期，发生于复位撤销之后。

4.3.10.1.2. Slave-Transmitter Operation for a Single Byte

When another I2C master device on the bus addresses the DW_apb_i2c and requests data, the DW_apb_i2c acts as a slave-transmitter and the following steps occur:

1. The other I2C master device initiates an I2C transfer with an address that matches the slave address in the [IC_SAR](#) register of the DW_apb_i2c.
2. The DW_apb_i2c acknowledges the sent address and recognizes the direction of the transfer to indicate that it is acting as a slave-transmitter.
3. The DW_apb_i2c asserts the RD_REQ interrupt (bit five of the [IC_RAW_INTR_STAT](#) register) and holds the [SCL](#) line low. It is in a wait state until software responds. If the RD_REQ interrupt has been masked, due to [IC_INTR_MASK.M_RD_REQ](#) being set to zero, then it is recommended that a hardware and/or software timing routine be used to instruct the CPU to perform periodic reads of the [IC_RAW_INTR_STAT](#) register.
 - a. Reads that indicate [IC_RAW_INTR_STAT.RD_REQ](#) being set to one must be treated as the equivalent of the RD_REQ interrupt being asserted.
 - b. Software must then act to satisfy the I2C transfer.
 - c. The timing interval used should be in the order of 10 times the fastest [SCL](#) clock period the DW_apb_i2c can handle. For example, for 400kbps, the timing interval is 25µs.

NOTE

The value of 10 is recommended here because this is approximately the amount of time required for a single byte of data transferred on the I2C bus.

1. If there is any data remaining in the Tx FIFO before receiving the read request, then the DW_apb_i2c asserts a TX_ABRT interrupt (bit six of the [IC_RAW_INTR_STAT](#) register) to flush the old data from the TX FIFO. If the TX_ABRT interrupt has been masked, due to [IC_INTR_MASK.M_TX_ABRT](#) being set to zero, then it is recommended that re-using the timing routine (described in the previous step), or a similar one, be used to read the [IC_RAW_INTR_STAT](#) register.

NOTE

Because the DW_apb_i2c's Tx FIFO is forced into a flushed/reset state whenever a TX_ABRT event occurs, it is necessary for software to release the DW_apb_i2c from this state by reading the [IC_CLR_TX_ABRT](#) register before attempting to write into the Tx FIFO. See register [IC_RAW_INTR_STAT](#) for more details.

- a. Reads that indicate bit six (R_TX_ABRT) being set to one must be treated as the equivalent of the TX_ABRT interrupt being asserted.
- b. There is no further action required from software.
- c. The timing interval used should be similar to that described in the previous step for the [IC_RAW_INTR_STAT.RD_REQ](#) register.
 1. Software writes to the [IC_DATA_CMD](#) register with the data to be written (by writing a '0' in bit 8).
 2. Software must clear the RD_REQ and TX_ABRT interrupts (bits five and six, respectively) of the [IC_RAW_INTR_STAT](#) register before proceeding. If the RD_REQ and/or TX_ABRT interrupts have been masked, then clearing of the [IC_RAW_INTR_STAT](#) register will have already been performed when either the R_RD_REQ or R_TX_ABRT bit has been read as one.
 3. The DW_apb_i2c releases the [SCL](#) and transmits the byte.
 4. The master may hold the I2C bus by issuing a RESTART condition or release the bus by issuing a STOP condition.

4.3.10.1.2. 单字节从设备发送操作

当总线上另一I2C主设备寻址DW_apb_i2c并请求数据时，DW_apb_i2c作为从设备发送器，执行以下步骤：

1. 另一I2C主设备发起的I2C传输，其地址与DW_apb_i2c的IC_SAR寄存器中设置的从设备地址匹配。
2. DW_apb_i2c确认所发送地址，并识别传输方向，表明其作为从设备发送器运行。
3. DW_apb_i2c触发RD_REQ中断（IC_RAW_INTR_STAT寄存器第5位），并将SCL线拉低，进入等待状态，直至软件响应。如果由于IC_INTR_MASK.M_RD_REQ被设置为零而导致RD_REQ中断被屏蔽，建议使用硬件和/或软件定时程序，指示CPU定期读取IC_RAW_INTR_STAT寄存器。
 - a. 将IC_RAW_INTR_STAT.RD_REQ设为一的读取操作必须视同RD_REQ中断被触发。
 - b. 软件随后必须采取措施以完成I2C传输。
 - c. 所使用的定时间隔应约为DW_apb_i2c可处理的最快SCL时钟周期的十倍。例如，对于400kbps，定时间隔为25μs。

i 注意

此处推荐使用数值10，原因是这大致相当于I2C总线上传输单个字节所需的时间。

1. 如果在收到读取请求之前，Tx FIFO中仍有数据残留，则DW_apb_i2c会触发TX_ABRT中断（IC_RAW_INTR_STAT寄存器的第六位）以清空Tx FIFO中的旧数据。若由于IC_INTR_MASK.M_TX_ABRT被设置为零而导致TX_ABRT中断被屏蔽，建议重复使用前述定时程序或类似程序，读取IC_RAW_INTR_STAT寄存器。

i 注意

由于每当发生TX_ABRT事件时，DW_apb_i2c的Tx FIFO会被强制刷新或重置，软件必须通过读取IC_CLR_TX_ABRT寄存器以释放DW_apb_i2c的该状态，然后方可尝试向Tx FIFO写入数据。详见寄存器IC_RAW_INTR_STAT。

- a. 当读取中第六位（R_TX_ABRT）被置为1时，应视为TX_ABRT中断已被断言。
- b. 软件无需采取进一步操作。
- c. 所使用的时间间隔应与前一步中对IC_RAW_INTR_STAT.RD_REQ寄存器描述的时间间隔类似。
 1. 软件通过向IC_DATA_CMD寄存器写入数据（将第8位写为'0'）实现写操作。
 2. 软件在继续执行前，必须清除IC_RAW_INTR_STAT寄存器中的RD_REQ和TX_ABRT中断（分别对应第5位和第6位）。如果RD_REQ和/或TX_ABRT中断已被屏蔽，则当读取到R_RD_REQ或R_TX_ABRT位为1时，系统将已完成对IC_RAW_INTR_STAT寄存器的清除。
 3. DW_apb_i2c释放SCL并传输字节。
 4. 主设备可通过发出RESTART条件保持I2C总线，或通过发出STOP条件释放总线。

NOTE

Slave-Transmitter Operation for a Single Byte is not applicable in Ultra-Fast Mode as Read transfers are not supported.

4.3.10.1.3. Slave-Receiver Operation for a Single Byte

When another I2C master device on the bus addresses the DW_apb_i2c and is sending data, the DW_apb_i2c acts as a slave-receiver and the following steps occur:

1. The other I2C master device initiates an I2C transfer with an address that matches the DW_apb_i2c's slave address in the [IC_SAR](#) register.
2. The DW_apb_i2c acknowledges the sent address and recognizes the direction of the transfer to indicate that the DW_apb_i2c is acting as a slave-receiver.
3. DW_apb_i2c receives the transmitted byte and places it in the receive buffer.

NOTE

If the Rx FIFO is completely filled with data when a byte is pushed, then the DW_apb_i2c slave holds the I2C [SCL](#) line low until the Rx FIFO has some space, and then continues with the next read request.

1. DW_apb_i2c asserts the RX_FULL interrupt [IC_RAW_INTR_STAT.RX_FULL](#). If the RX_FULL interrupt has been masked, due to setting [IC_INTR_MASK.M_RX_FULL](#) register to zero or setting [IC_TX_TL](#) to a value larger than zero, then it is recommended that a timing routine (described in [Section 4.3.10.1.2](#)) be implemented for periodic reads of the [IC_STATUS](#) register. Reads of the [IC_STATUS](#) register, with bit 3 (RFNE) set at one, must then be treated by software as the equivalent of the RX_FULL interrupt being asserted.
2. Software may read the byte from the [IC_DATA_CMD](#) register (bits 7:0).
3. The other master device may hold the I2C bus by issuing a RESTART condition, or release the bus by issuing a STOP condition.

4.3.10.1.4. Slave-Transfer Operation For Bulk Transfers

In the standard I2C protocol, all transactions are single byte transactions and the programmer responds to a remote master read request by writing one byte into the slave's TX FIFO. When a slave (slave-transmitter) is issued with a read request (RD_REQ) from the remote master (master-receiver), at a minimum there should be at least one entry placed into the slave-transmitter's TX FIFO. DW_apb_i2c is designed to handle more data in the TX FIFO so that subsequent read requests can take that data without raising an interrupt to get more data. Ultimately, this eliminates the possibility of significant latencies being incurred between raising the interrupt for data each time had there been a restriction of having only one entry placed in the TX FIFO. This mode only occurs when DW_apb_i2c is acting as a slave-transmitter. If the remote master acknowledges the data sent by the slave-transmitter and there is no data in the slave's TX FIFO, the DW_apb_i2c holds the I2C [SCL](#) line low while it raises the read request interrupt (RD_REQ) and waits for data to be written into the TX FIFO before it can be sent to the remote master.

If the RD_REQ interrupt is masked, due to [IC_INTR_STAT.R_RD_REQ](#) set to zero, then it is recommended that a timing routine be used to activate periodic reads of the [IC_RAW_INTR_STAT](#) register. Reads of [IC_RAW_INTR_STAT](#) that return bit five (RD_REQ) set to one must be treated as the equivalent of the RD_REQ interrupt referred to in this section. This timing routine is similar to that described in [Section 4.3.10.1.2](#).

The RD_REQ interrupt is raised upon a read request, and like interrupts, must be cleared when exiting the interrupt service handling routine (ISR). The ISR allows you to either write one byte or more than one byte into the Tx FIFO. During the transmission of these bytes to the master, if the master acknowledges the last byte, then the slave must raise the RD_REQ again because the master is requesting for more data. If the programmer knows in advance that the remote master is requesting a packet of 'n' bytes, then when another master addresses DW_apb_i2c and requests data, the Tx FIFO could be written with 'n' bytes and the remote master receives it as a continuous stream of data. For example, the

注意

单字节从机发送操作在超高速模式下不适用，因为该模式不支持读取传输。

4.3.10.1.3 单字节从机接收操作

当总线上另一I2C主设备寻址DW_apb_i2c并发送数据时，DW_apb_i2c作为从机接收者，执行以下步骤：

1. 另一I2C主设备使用与DW_apb_i2c在IC_SAR寄存器中所设从机地址匹配的地址启动I2C传输。
2. DW_apb_i2c确认所发送的地址，并识别传输方向，从而表明其作为从机接收者。
3. DW_apb_i2c接收传输的字节并存入接收缓冲区。

注意

如果在推送字节时，Rx FIFO 已完全填满数据，则 DW_apb_i2c 从机会将 I2C SCL 线拉低，直到 Rx FIFO 有可用空间，然后继续执行下一个读取请求。

1. DW_apb_i2c 会触发 RX_FULL 中断 IC_RAW_INTR_STAT.RX_FULL。若因将 IC_INTR_MASK.M_RX_FULL 寄存器设为零或将 IC_TX_TL 设为大于零的值而屏蔽了 RX_FULL 中断，建议实现一个时序例程（详见第 4.3.10.1.2 节）以周期性读取 IC_STATUS 寄存器。软件读取 IC_STATUS 寄存器时，若第 3 位 (RFNE) 为 1，应将该状态视为 RX_FULL 中断已被触发的等效状态。
2. 软件可从 IC_DATA_CMD 寄存器 (位 7:0) 读取该字节。
3. 另一主设备可能通过发出 RESTART 条件来占用 I2C 总线，或通过发出 STOP 条件释放总线。

4.3.10.1.4. 大批量传输的从设备传输操作

在标准I2C协议中，所有事务均为单字节传输，程序通过向从设备的TX FIFO写入一个字节以响应远程主机的读取请求。当从设备（从机发送器）收到远程主机（主机接收器）发出的读取请求 (RD_REQ) 时，TX FIFO中至少应存放一条数据。DW_apb_i2c设计支持在TX FIFO中存储更多数据，以使后续读取请求能够获取这些数据，而无需触发中断以获取更多数据。此举最终消除了若仅允许TX FIFO中存放一条数据而每次获取数据均触发中断所可能导致的显著延迟。此模式仅在DW_apb_i2c作为从机发送器时发生。若远程主机确认了从机发送器发送的数据且从机的TX FIFO中无数据，DW_apb_i2c将在触发读取请求中断 (RD_REQ) 时将I2C SCL线维持低电平，并等待数据写入TX FIFO后方能发送至远程主机。

如果由于 IC_INTR_STAT.R_RD_REQ 被置零而导致 RD_REQ 中断被屏蔽，建议使用定时例程周期性激活对 IC_RAW_INTR_STAT 寄存器的读取。对 IC_RAW_INTR_STAT 的读取返回第5位 (RD_REQ) 为1时，必须视同本节所述的 RD_REQ 中断。该定时例程与第4.3.10.1.2节中描述的类似。

RD_REQ 中断在读请求发生时触发，且须在中断服务例程 (ISR) 退出时清除，如同其他中断。ISR 允许写入一个或多个字节至 Tx FIFO。在向主设备传输这些字节期间，若主设备确认了最后一个字节，则从设备必须再次触发 RD_REQ，因为主设备请求更多数据。若程序员事先已知远程主机请求 ‘n’ 字节的数据包，则当其他主机访问 DW_apb_i2c 并请求数据时，可向 Tx FIFO 写入 ‘n’ 字节，远程主机会连续接收该数据流。例如，

DW_apb_i2c slave continues to send data to the remote master as long as the remote master is acknowledging the data sent and there is data available in the Tx FIFO. There is no need to hold the **SCL** line low or to issue RD_REQ again.

If the remote master is to receive 'n' bytes from the DW_apb_i2c but the programmer wrote a number of bytes larger than 'n' to the Tx FIFO, then when the slave finishes sending the requested 'n' bytes, it clears the Tx FIFO and ignores any excess bytes.

The DW_apb_i2c generates a transmit abort (TX_ABRT) event to indicate the clearing of the Tx FIFO in this example. At the time an ACK/NACK is expected, if a NACK is received, then the remote master has all the data it wants. At this time, a flag is raised within the slave's state machine to clear the leftover data in the Tx FIFO. This flag is transferred to the processor bus clock domain where the FIFO exists and the contents of the Tx FIFO is cleared at that time.

4.3.10.2. Master Mode Operation

This section discusses master mode procedures.

4.3.10.2.1. Initial Configuration

To use the DW_apb_i2c as a master perform the following steps:

1. Disable the DW_apb_i2c by writing zero to **IC_ENABLE**.ENABLE.
2. Write to the **IC_CON** register to set the maximum speed mode supported (bits 2:1) and the desired speed of the DW_apb_i2c master-initiated transfers, either 7-bit or 10-bit addressing (bit 4). Ensure that bit six (**IC_SLAVE_DISABLE**) is written with a '1' and bit zero (**MASTER_MODE**) is written with a '1'.

Note: Slaves and masters do not have to be programmed with the same type of 7-bit or 10-bit address. For instance, a slave can be programmed with 7-bit addressing and a master with 10-bit addressing, and vice versa.

1. Write to the **IC_TAR** register the address of the I2C device to be addressed (bits 9:0). This register also indicates whether a General Call or a START BYTE command is going to be performed by I2C.
2. Enable the DW_apb_i2c by writing a one to **IC_ENABLE**.ENABLE.
3. Now write transfer direction and data to be sent to the **IC_DATA_CMD** register. If the **IC_DATA_CMD** register is written before the DW_apb_i2c is enabled, the data and commands are lost as the buffers are kept cleared when DW_apb_i2c is disabled. This step generates the START condition and the address byte on the DW_apb_i2c. Once DW_apb_i2c is enabled and there is data in the TX FIFO, DW_apb_i2c starts reading the data.

i **NOTE**

Depending on the reset values chosen, steps two, three, four, and five may not be necessary because the reset values can be configured. The values stored are static and do not need to be reprogrammed if the DW_apb_i2c is disabled, with the exception of the transfer direction and data.

4.3.10.2.2. Master Transmit and Master Receive

The DW_apb_i2c supports switching back and forth between reading and writing dynamically. To transmit data, write the data to be written to the lower byte of the I2C Rx/Tx Data Buffer and Command Register (**IC_DATA_CMD**). The CMD bit [8] should be written to zero for I2C write operations. Subsequently, a read command may be issued by writing "don't cares" to the lower byte of the **IC_DATA_CMD** register, and a one should be written to the CMD bit. The DW_apb_i2c master continues to initiate transfers as long as there are commands present in the transmit FIFO. If the transmit FIFO becomes empty the master either inserts a STOP condition after completing the current transfers.

- If set to one, it issues a STOP condition after completing the current transfer.
- If set to zero, it holds **SCL** low until next command is written to the transmit FIFO.

For more details, refer to [Section 4.3.7](#).

只要远程主控确认已接收所发送的数据且Tx FIFO中仍有可用数据，DW_apb_i2c从设备将持续向远程主控发送数据，无需将SCL线保持为低电平或再次发出RD_REQ信号。

若远程主控预期从DW_apb_i2c接收‘n’字节数据，但程序员向Tx FIFO写入字节数大于‘n’，则从设备在发送完请求的‘n’字节后，会清空Tx FIFO并忽略多余字节。

在此示例中，DW_apb_i2c会产生传输中止（TX_ABRT）事件，以指示已清空Tx FIFO。当预期接收ACK/NACK时，若接收到NACK，则表明远程主控已获取全部所需数据。此时，从设备状态机内部的标志位被激活，用于清除Tx FIFO中剩余的数据。该标志位会传递至包含该FIFO的处理器总线时钟域，在该时钟域中执行清空操作。

4.3.10.2. 主机模式操作

本节讨论主机模式的操作流程。

4.3.10.2.1. 初始配置

若需将DW_apb_i2c用作主机，请执行以下步骤：

1. 通过向IC_ENABLE.ENABLE写入零以禁用DW_apb_i2c。
2. 写入IC_CON寄存器，设置支持的最高速度模式（第2至1位）及所需的DW_apb_i2c主机启动传输速度，同时设置7位或10位寻址模式（第4位）。确保第六位（IC_SLAVE_DISABLE）写入‘1’，第零位（MASTER_MODE）写入‘1’。

注意：从机与主机的地址类型（7位或10位）不必相同。例如，主机可设置为10位寻址，从机可设置为7位寻址，反之亦然。

1. 向IC_TAR寄存器写入要寻址的I2C设备地址（第9至0位）。该寄存器还指示I2C是否将执行通用调用（General Call）或启动字节（START BYTE）命令。
2. 通过向IC_ENABLE.ENABLE位写入1来启用DW_apb_i2c。
3. 现在将传输方向和要发送的数据写入IC_DATA_CMD寄存器。如果在启用DW_apb_i2c之前写入IC_DATA_CMD寄存器，则数据和命令将丢失，因DW_apb_i2c禁用时缓冲区会保持清空状态。此步骤在DW_apb_i2c上产生启动条件和地址字节。一旦启用DW_apb_i2c且TX FIFO中有数据，DW_apb_i2c便开始读取数据。

注意

根据选择的复位值，步骤二、三、四和五可能不必执行，因为复位值可被配置。所存储的值为静态值，除传输方向和数据外，即使DW_apb_i2c被禁用也无需重新编程。

4.3.10.2.2. 主设备发送与主设备接收

DW_apb_i2c 支持在读写之间动态切换。要传输数据，请将待写入的数据写入 I2C 接收/发送数据缓冲区和命令寄存器（IC_DATA_CMD）的低字节。对于 I2C 写操作，CMD 位[8]应写为零。随后，可通过向 IC_DATA_CMD 寄存器低字节写入“无关数据”，并将 CMD 位写为一来发出读命令。只要发送 FIFO 中存在命令，DW_apb_i2c 主设备将持续发起传输。若发送 FIFO 变为空，主设备将在完成当前传输后插入停止（STOP）条件。

- 若设置为 1，则在完成当前传输后发出停止（STOP）条件。
- 若设置为 0，则保持 SCL 线处于低电平，直到下一条命令写入发送 FIFO。

详细信息请参见第4.3.7节。

4.3.10.3. Disabling DW_apb_i2c

The register [IC_ENABLE_STATUS](#) is added to allow software to unambiguously determine when the hardware has completely shutdown in response to [IC_ENABLE](#).ENABLE being set from one to zero.

Only one register is required to be monitored, as opposed to monitoring two registers ([IC_STATUS](#) and [IC_RAW_INTR_STAT](#)) which was a requirement for earlier versions of DW_apb_i2c.

i NOTE

The DW_apb_i2c Master can be disabled only if the current command being processed—when the ic_enable deassertion occurs—has the STOP bit set to one. When an attempt is made to disable the DW_apb_i2c Master while processing a command without the STOP bit set, the DW_apb_i2c Master continues to remain active, holding the [SCL](#) line low until a new command is received in the Tx FIFO. When the DW_apb_i2c Master is processing a command without the STOP bit set, you can issue the ABORT ([IC_ENABLE](#).ABORT) to relinquish the I2C bus and then disable DW_apb_i2c.

4.3.10.3.1. Procedure

1. Define a timer interval (t_{i2c_poll}) equal to the 10 times the signalling period for the highest I2C transfer speed used in the system and supported by DW_apb_i2c. For example, if the highest I2C transfer mode is 400kbps, then this t_{i2c_poll} is 25μs.
2. Define a maximum time-out parameter, MAX_T_POLL_COUNT, such that if any repeated polling operation exceeds this maximum value, an error is reported.
3. Execute a blocking thread/process/function that prevents any further I2C master transactions to be started by software, but allows any pending transfers to be completed.

i NOTE

This step can be ignored if DW_apb_i2c is programmed to operate as an I2C slave only.

1. The variable POLL_COUNT is initialized to zero.
2. Set bit zero of the [IC_ENABLE](#) register to zero.
3. Read the [IC_ENABLE_STATUS](#) register and test the IC_EN bit (bit 0). Increment POLL_COUNT by one. If POLL_COUNT >= MAX_T_POLL_COUNT, exit with the relevant error code.
4. If [IC_ENABLE_STATUS](#)[0] is one, then sleep for t_{i2c_poll} and proceed to the previous step. Otherwise, exit with a relevant success code.

4.3.10.4. Aborting I2C Transfers

The ABORT control bit of the [IC_ENABLE](#) register allows the software to relinquish the I2C bus before completing the issued transfer commands from the Tx FIFO. In response to an ABORT request, the controller issues the STOP condition over the I2C bus, followed by Tx FIFO flush. Aborting the transfer is allowed only in master mode of operation.

4.3.10.4.1. Procedure

1. Stop filling the Tx FIFO ([IC_DATA_CMD](#)) with new commands.
2. When operating in DMA mode, disable the transmit DMA by setting TDMAE to zero.
3. Set [IC_ENABLE](#).ABORT to one.
4. Wait for the M_TX_ABRT interrupt.

4.3.10.3. 禁用 DW_apb_i2c

添加了寄存器 IC_ENABLE_STATUS，使软件能够明确判断硬件在 IC_ENABLE.ENABLE 从 1 变为 0 后何时完全关闭。

只需监控一个寄存器，而非早期版本 DW_apb_i2c 需要监控的两个寄存器（IC_STATUS 和 IC_RAW_INTR_STAT）。

i 注意

只有当 ic_enable 取消断言时，当前正在处理的命令的 STOP 位被置为 1，DW_apb_i2c 主控器才能被禁用。若尝试在处理未设置 STOP 位的命令时禁用 DW_apb_i2c 主控器，则主控器将保持激活状态，并保持 SCL 线路为低电平，直到 Tx FIFO 中接收到新的命令。处理未设置 STOP 位的命令时，可以通过发出 ABORT (IC_ENABLE.ABORT) 指令释放 I2C 总线，然后禁用 DW_apb_i2c。

4.3.10.3.1. 操作流程

1. 定义一个计时器间隔 (t_{i2c_poll})，其等于系统中使用且由DW_apb_i2c支持的最高I2C传输速率的信号周期的10倍。例如，若最高I2C传输模式为400kbps，则 t_{i2c_poll} 为25μs。
2. 定义最大超时参数MAX_T_POLL_COUNT，若任何重复轮询操作超过该最大值，则报告错误。
3. 执行阻塞线程/进程/函数，阻止软件启动任何新的I2C主设备事务，但允许正在进行的传输完成。

i 注意

若DW_apb_i2c仅配置为I2C从机模式，此步骤可忽略。

1. 变量POLL_COUNT初始化为零。
2. 将IC_ENABLE寄存器的第0位清零。
3. 读取IC_ENABLE_STATUS寄存器，并检测IC_EN位（第0位）。将 POLL_COUNT 加一。若 POLL_COUNT >= MAX_T_POLL_COUNT，则返回相应错误代码并退出。
4. 若 IC_ENABLE_STATUS[0] 为 1，则休眠 t_{i2c_poll} 并返回至前一步。否则，返回相应成功代码并退出。

4.3.10.4. 中止 I2C 传输

IC_ENABLE 寄存器中的 ABORT 控制位允许软件在未完成从 Tx FIFO 发出的传输命令前释放 I2C 总线。接收到 ABORT 请求时，控制器会在 I2C 总线上发出 STOP 条件，随后执行 Tx FIFO 清空。仅允许运行于主机模式时中止传输。

4.3.10.4.1. 操作步骤

1. 停止向 Tx FIFO (IC_DATA_CMD) 写入新命令。
2. 在 DMA 模式下，将 TDMAE 清零以禁用发送 DMA。
3. 将 IC_ENABLE.ABORT 置为 1。
4. 等待 M_TX_ABRT 中断信号。

5. Read the `IC_TX_ABRT_SOURCE` register to identify the source as ABRT_USER_ABRT.

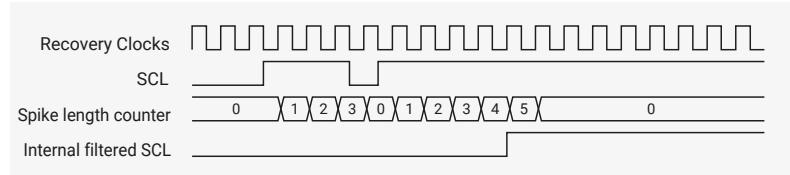
4.3.11. Spike Suppression

The DW_apb_i2c contains programmable spike suppression logic that match requirements imposed by the I2C Bus Specification for SS/FS modes. This logic is based on counters that monitor the input signals (`SCL` and `SDA`), checking if they remain stable for a predetermined amount of `ic_clk` cycles before they are sampled internally. There is one separate counter for each signal (`SCL` and `SDA`). The number of `ic_clk` cycles can be programmed by the user and should be calculated taking into account the frequency of `ic_clk` and the relevant spike length specification. Each counter is started whenever its input signal changes its value. Depending on the behaviour of the input signal, one of the following scenarios occurs:

- The input signal remains unchanged until the counter reaches its count limit value. When this happens, the internal version of the signal is updated with the input value, and the counter is reset and stopped. The counter is not restarted until a new change on the input signal is detected.
- The input signal changes again before the counter reaches its count limit value. When this happens, the counter is reset and stopped, but the internal version of the signal is not updated. The counter remains stopped until a new change on the input signal is detected.

The timing diagram in [Figure 83](#) illustrates the behaviour described above.

Figure 83. Spike Suppression Example



❶ NOTE

There is a 2-stage synchronizer on the `SCL` input, but for the sake of simplicity this synchronization delay was not included in the timing diagram in [Figure 83](#).

The I2C Bus Specification calls for different maximum spike lengths according to the operating mode – 50ns for SS and FS, so this register is required to store the values needed:

- Register `IC_FS_SPKLEN` holds the maximum spike length for SS and FS modes

This register is 8 bits wide and accessible through the APB interface for read and write purposes; however, they can be written to only when the DW_apb_i2c is disabled. The minimum value that can be programmed into these registers is one; attempting to program a value smaller than one results in the value one being written.

The default value for these registers is based on the value of 100ns for `ic_clk` period, so should be updated for the `clk_sys` period in use on RP2040.

5. 读取 IC_TX_ABRT_SOURCE 寄存器以识别源为 ABRT_USER_ABRT。

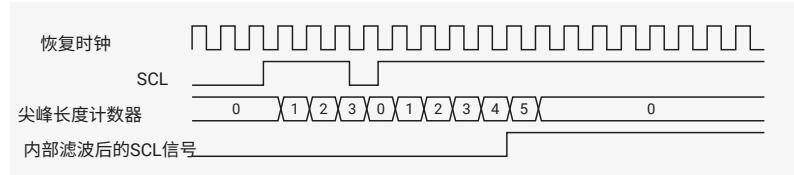
4.3.11. 尖峰抑制

DW_apb_i2c 包含可编程的脉冲抑制逻辑，以满足 I2C 总线规范中对 SS/FS 模式的要求。该逻辑基于计数器，监测输入信号（SCL 和 SDA），核查其在采样前是否在预定数量的 `ic_clk` 周期内保持稳定。每个信号（SCL 和 SDA）均配备独立的计数器。计数的 `ic_clk` 周期数由用户可编程，并应根据 `ic_clk` 频率及相关脉冲宽度规范计算确定。每当输入信号的值发生变化时，相应的计数器即被启动。依据输入信号的行为，将出现以下情形之一：

- 输入信号保持不变，直至计数器达到其计数上限值。此时，内部信号值更新为输入值，计数器复位并停止。计数器在检测到输入信号的新变化之前不会重新启动。
- 计数器达到计数限制值之前，输入信号已再次发生变化。出现此情况时，计数器被复位并停止，但信号的内部版本未更新。计数器将保持停止状态，直至检测到输入信号的新变化。

图83中的时序图说明了上述行为。

图83。尖峰抑制示例



① 注意

SCL输入端设有两级同步器，但为简化起见，图83的时序图未包含该同步延迟。

I2C总线规范根据工作模式对最大尖峰长度有不同要求——SS和FS模式为50纳秒，因此该寄存器用于存储所需值：

- 寄存器IC_FS_SPKLEN保存SS和FS模式下的最大尖峰长度值

该寄存器位宽为8位，可通过APB接口进行读写访问；然而，仅当 DW_apb_i2c 被禁用时，才允许写入这些寄存器。这些寄存器可编程的最小值为 1；尝试编程小于 1 的值时，实际写入的值为 1。

这些寄存器的默认值基于 `ic_clk` 周期为 100ns，应根据 RP2040 上使用的 `clk_sys` 周期进行更新。

NOTE

- Because the minimum value that can be programmed into the `IC_FS_SPKLEN` register is one, the spike length specification can be exceeded for low frequencies of `ic_clk`. Consider the simple example of a 10MHz (100ns period) `ic_clk`; in this case, the minimum spike length that can be programmed is 100ns, which means that spikes up to this length are suppressed.
- Standard synchronization logic (two flip-flops in series) is implemented upstream of the spike suppression logic and is not affected in any way by the contents of the spike length registers or the operation of the spike suppression logic; the two operations (synchronization and spike suppression) are completely independent. Because the `SCL` and `SDA` inputs are asynchronous to `ic_clk`, there is one `ic_clk` cycle uncertainty in the sampling of these signals; that is, depending on when they occur relative to the rising edge of `ic_clk`, spikes of the same original length might show a difference of one `ic_clk` cycle after being sampled.
- Spike suppression is symmetrical; that is, the behaviour is exactly the same for transitions from zero to one and from one to zero.

4.3.12. Fast Mode Plus Operation

In fast mode plus, the DW_apb_i2c allows the fast mode operation to be extended to support speeds up to 1000kbps. To enable the DW_apb_i2c for fast mode plus operation, perform the following steps before initiating any data transfer:

- Set `ic_clk` frequency greater than or equal to 32MHz (refer to [Section 4.3.14.2.1](#)).
- Program the `IC_CON` register [2:1] = 2'b10 for fast mode or fast mode plus.
- Program `IC_FS_SCL_LCNT` and `IC_FS_SCL_HCNT` registers to meet the fast mode plus `SCL` (refer to [Section 4.3.14](#)).
- Program the `IC_FS_SPKLEN` register to suppress the maximum spike of 50ns.
- Program the `IC_SDA_SETUP` register to meet the minimum data setup time (`tSU; DAT`).

4.3.13. Bus Clear Feature

DW_apb_i2c supports the bus clear feature that provides graceful recovery of data `SDA` and clock `SCL` lines during unlikely events in which either the clock or data line is stuck at LOW.

4.3.13.1. SDA Line Stuck at LOW Recovery

In case of `SDA` line stuck at LOW, the master performs the following actions to recover as shown in [Figure 84](#) and [Figure 85](#):

- Master sends a maximum of nine clock pulses to recover the bus LOW within those nine clocks.
 - The number of clock pulses will vary with the number of bits that remain to be sent by the slave. As the maximum number of bits is nine, master sends up to nine clock pulses and allows the slave to recover it.
 - The master attempts to assert a Logic 1 on the `SDA` line and check whether `SDA` is recovered. If the `SDA` is not recovered, it will continue to send a maximum of nine `SCL` clocks.
- If `SDA` line is recovered within nine clock pulses then the master will send the STOP to release the bus.
- If `SDA` line is not recovered even after the ninth clock pulse then system needs a hardware reset.

注意

- 由于 IC_FS_SPKLEN 寄存器的最小可编程值为 1，低频 ic_clk 下尖峰长度规格可能被超出。以 ic_clk 为 10MHz（周期为 100ns）为例；在此情况下，可编程的最小尖峰长度为 100ns，因此可抑制最长达此长度的尖峰。
- 标准同步逻辑（两个串联触发器）位于尖峰抑制逻辑的上游，其不受尖峰长度寄存器内容或尖峰抑制逻辑操作的任何影响；这两个操作（同步和脉冲抑制）完全独立。
由于 SCL 和 SDA 输入信号相对于 ic_clk 是异步的，对这些信号的采样存在一个 ic_clk 周期的不确定性；即根据它们相对于 ic_clk 上升沿发生的时间不同，采样后同一原始长度的脉冲可能出现一个 ic_clk 周期的差异。
- 脉冲抑制是对称的；即从零跳变到一和从一跳变到零的行为完全相同。

4.3.12. 快速模式 Plus 操作

在快速模式加（fast mode plus）下，DW_apb_i2c 允许将快速模式操作扩展至支持高达 1000kbps 的速率。要启用 DW_apb_i2c 的快速模式加操作，请在启动任何数据传输前完成以下步骤：

- 将 ic_clk 频率设置为大于或等于 32MHz（参见第 4.3.14.2.1 节）。
- 将 IC_CON 寄存器[2:1]编程为 2'b10，以支持快速模式或快速模式加。
- 编程 IC_FS_SCL_LCNT 和 IC_FS_SCL_HCNT 寄存器，以满足快速模式加 SCL 的要求（参见第 4.3.14 节）。
- 编程 IC_FS_SPKLEN 寄存器，以抑制最大 50 纳秒的尖峰脉冲。
- 编程 IC_SDA_SETUP 寄存器，以满足最小数据建立时间（tSU; DAT）要求。

4.3.13. 总线清除功能

DW_apb_i2c 支持总线清除功能，该功能能够在时钟线或数据线异常被拉低的少见情况下，优雅地恢复数据 SDA 和时钟 SCL 线。

4.3.13.1. SDA 线异常被拉低的恢复

若 SDA 线异常拉低，主设备将执行下述操作进行恢复，如图 84 和图 85 所示：

- 主设备发送最多九个时钟脉冲，以在该九个时钟周期内恢复总线低电平状态。
 - 时钟脉冲的数量将随从设备剩余待发送位数的变化而变化。由于最大位数为九位，主设备发送最多九个时钟脉冲，并允许从设备进行恢复。
 - 主设备尝试在 SDA 线上施加逻辑 1，并检测 SDA 是否被恢复。如果 SDA 未被恢复，主设备将继续发送最多九个 SCL 时钟脉冲。
- 若 SDA 线在九个时钟脉冲内恢复，主设备将发送停止信号以释放总线。
- 若在第九个时钟脉冲后 SDA 线仍未恢复，系统需执行硬件复位。

Figure 84. SDA Recovery with 9 SCL Clocks

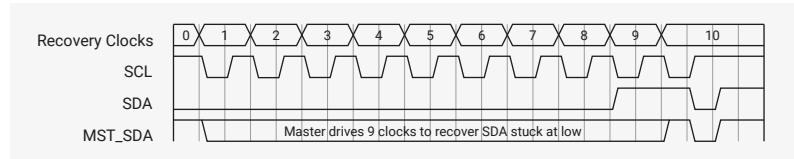
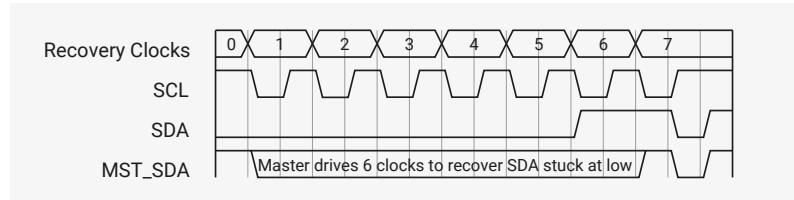


Figure 85. SDA Recovery with 6 SCL Clocks



4.3.13.2. SCL Line is Stuck at LOW

In the unlikely event (due to an electric failure of a circuit) where the clock (**SCL**) is stuck to LOW, there is no effective method to overcome this problem but to reset the bus using the hardware reset signal.

4.3.14. IC_CLK Frequency Configuration

When the DW_apb_i2c is configured as a Standard (SS), Fast (FS)/Fast-Mode Plus (FM+), the *CNT registers must be set before any I2C bus transaction can take place in order to ensure proper I/O timing. The *CNT registers are:

- [IC_SS_SCL_HCNT](#)
- [IC_SS_SCL_LCNT](#)
- [IC_FS_SCL_HCNT](#)
- [IC_FS_SCL_LCNT](#)

i NOTE

The tBUF timing and setup/hold time of START, STOP and RESTART registers uses *HCNT/*LCNT register settings for the corresponding speed mode.

i NOTE

It is not necessary to program any of the *CNT registers if the DW_apb_i2c is enabled to operate only as an I2C slave, since these registers are used only to determine the **SCL** timing requirements for operation as an I2C master.

Table 449 lists the derivation of I2C timing parameters from the *CNT programming registers.

Table 449. Derivation of I2C Timing Parameters from *CNT Registers

Timing Parameter	Symbol	Standard Speed	Fast Speed / Fast Speed Plus
LOW period of the SCL clock	tLOW	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
HIGH period of the SCL clock	tHIGH	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
Setup time for a repeated START condition	tSU;STA	IC_SS_SCL_LCNT	IC_FS_SCL_HCNT
Hold time (repeated) START condition*	tHD;STA	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
Setup time for STOP condition	tSU;STO	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT

图84. SDA
使用9个 **SCL** 时钟进行恢复时钟

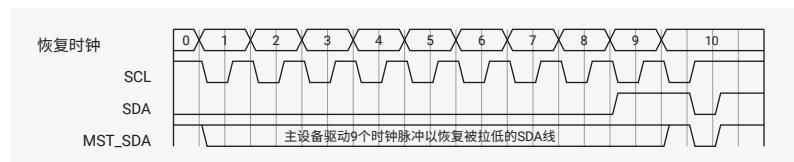
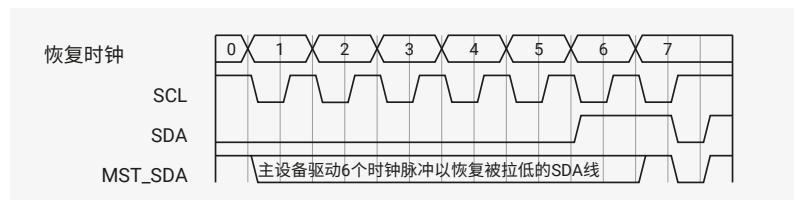


图85. SDA
使用6个 **SCL** 时钟进行恢复时钟



4.3.13.2. **SCL** 线被长时间拉低

在极少发生的情况下（由于电路电气故障），若时钟（**SCL**）保持低电平且无法恢复，则唯一有效的解决方法是通过硬件复位信号重置总线。

4.3.14. **IC_CLK** 频率配置

当DW_apb_i2c配置为标准模式（SS）、快速模式（FS）或快速增强模式（FM+）时，必须在任何I2C总线事务开始前设置*CNT寄存器，以确保正确的I/O时序。*CNT寄存器包括：

- [IC_SS_SCL_HCNT](#)
- [IC_SS_SCL_LCNT](#)
- [IC_FS_SCL_HCNT](#)
- [IC_FS_SCL_LCNT](#)

注意

tBUF时序及START、STOP和RESTART寄存器的建立/保持时间均采用对应速度模式的*HCNT/*LCNT寄存器设置。

注意

若DW_apb_i2c仅配置为I2C从设备运行，则无需配置任何*CNT寄存器，因这些寄存器仅用于确定I2C主设备模式下 **SCL** 的时序要求。

表449列出了基于*CNT编程寄存器推导I2C时序参数的方法。

表449。从*CNT寄存器推导I2C时序参数

时序参数	符号	标准速率	快速速率 / 快速速率增强
SCL 时钟的低电平周期	tLOW	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
SCL 时钟的高电平周期	tHIGH	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
重复启动条件的建立时间	tSU;STA	IC_SS_SCL_LCNT	IC_FS_SCL_HCNT
保持时间（重复）启动条件*	tHD;STA	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT
停止条件的建立时间	tSU;STO	IC_SS_SCL_HCNT	IC_FS_SCL_HCNT

Timing Parameter	Symbol	Standard Speed	Fast Speed / Fast Speed Plus
Bus free time between a STOP and a START condition	tBUF	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
Spike length	tSP	IC_FS_SPKLEN	IC_FS_SPKLEN
Data hold time	tHD;DAT	IC_SDA_HOLD	IC_SDA_HOLD
Data setup time	tSU;DAT	IC_SDA_SETUP	IC_SDA_SETUP

4.3.14.1. Minimum High and Low Counts in SS, FS, and FM+ Modes.

When the DW_apb_i2c operates as an I2C master, in both transmit and receive transfers:

- IC_SS_SCL_LCNT and IC_FS_SCL_LCNT register values must be larger than IC_FS_SPKLEN + 7.
- IC_SS_SCL_HCNT and IC_FS_SCL_HCNT register values must be larger than IC_FS_SPKLEN + 5.

Details regarding the DW_apb_i2c high and low counts are as follows:

- The minimum value of IC_*_SPKLEN + 7 for the *_LCNT registers is due to the time required for the DW_apb_i2c to drive SDA after a negative edge of SCL.
- The minimum value of IC_*_SPKLEN + 5 for the *_HCNT registers is due to the time required for the DW_apb_i2c to sample SDA during the high period of SCL.
- The DW_apb_i2c adds one cycle to the programmed *_LCNT value in order to generate the low period of the SCL clock; this is due to the counting logic for SCL low counting to (*_LCNT + 1).
- The DW_apb_i2c adds IC_*_SPKLEN + 7 cycles to the programmed *_HCNT value in order to generate the high period of the SCL clock; this is due to the following factors:
 - The counting logic for SCL high counts to (*_HCNT+1).
 - The digital filtering applied to the SCL line incurs a delay of SPKLEN + 2 ic_clk cycles, where SPKLEN is:
 - IC_FS_SPKLEN if the component is operating in SS or FS
 - Whenever SCL is driven one to zero by the DW_apb_i2c—that is, completing the SCL high time—an internal logic latency of three ic_clk cycles is incurred. Consequently, the minimum SCL low time of which the DW_apb_i2c is capable is nine ic_clk periods (7 + 1 + 1), while the minimum SCL high time is thirteen ic_clk periods (6 + 1 + 3 + 3).

i NOTE

The total high time and low time of SCL generated by the DW_apb_i2c master is also influenced by the rise time and fall time of the SCL line, as shown in the illustration and equations in Figure 86. It should be noted that the SCL rise and fall time parameters vary, depending on external factors such as:

- Characteristics of IO driver
- Pull-up resistor value
- Total capacitance on SCL line, and so on

These characteristics are beyond the control of the DW_apb_i2c.

时序参数	符号	标准速率	快速速率 / 快速速率增强
STOP与START条件之间的总线空闲时间	tBUF	IC_SS_SCL_LCNT	IC_FS_SCL_LCNT
脉冲宽度	tSP	IC_FS_SPKLEN	IC_FS_SPKLEN
数据保持时间	tHD;DAT	IC_SDA_HOLD	IC_SDA_HOLD
数据建立时间	tSU;DAT	IC_SDA_SETUP	IC_SDA_SETUP

4.3.14.1. SS、FS 及 FM+ 模式下的最小高低计数

当 DW_apb_i2c 作为 I2C 主设备运行时，在发送和接收传输过程中：

- IC_SS_SCL_LCNT 与 IC_FS_SCL_LCNT 寄存器值必须大于 IC_FS_SPKLEN + 7。
- IC_SS_SCL_HCNT 与 IC_FS_SCL_HCNT 寄存器值必须大于 IC_FS_SPKLEN + 5。

有关 DW_apb_i2c 高低计数的详细说明如下：

- *_LCNT 寄存器的最小值为 IC_*_SPKLEN + 7，该数值基于 DW_apb_i2c 在 SCL 下降沿后驱动 SDA 所需时间。
- *_HCNT 寄存器的最小值为 IC_*_SPKLEN + 5，该数值基于 DW_apb_i2c 在 SCL 高电平期间采样 SDA 所需时间。
- DW_apb_i2c 在编程的 *_LCNT 值基础上增加一个周期，以产生 SCL 时钟的低电平周期；这是因为 SCL 低电平计数逻辑计数至 (*_LCNT + 1)。
- DW_apb_i2c 在编程的 *_HCNT 值基础上增加 IC_*_SPKLEN + 7 个周期，以产生 SCL 时钟的高电平周期；其原因如下：
 - SCL 高电平计数逻辑计数至 (*_HCNT + 1)。
 - 施加于 SCL 线的数字滤波导致延迟为 SPKLEN + 2 个 ic_clk 周期，其中 SPKLEN 定义为：
 - 当组件工作于 SS 或 FS 模式时，为 IC_FS_SPKLEN
 - 每当 DW_apb_i2c 将 SCL 信号由高拉低——即完成 SCL 的高电平周期时——内部逻辑会产生三个 ic_clk 周期的延迟。因此，DW_apb_i2c 所能支持的最小 SCL 低电平时间为九个 ic_clk 周期 (7 + 1 + 1)，而最小 SCL 高电平时间为十三个 ic_clk 周期 (6 + 1 + 3) + 3)。

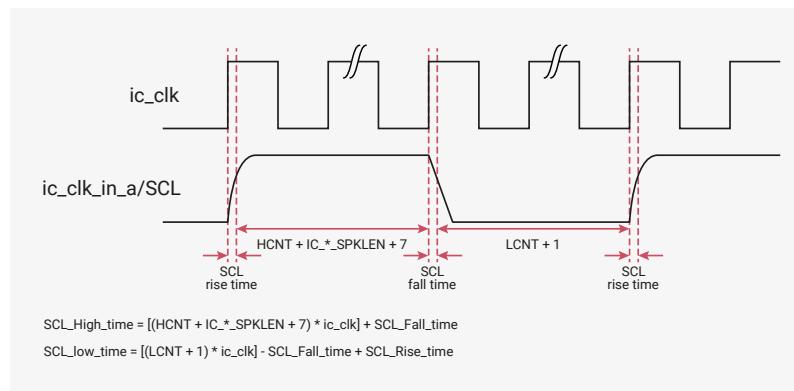
i 注意

DW_apb_i2c 主设备产生的 SCL 总高电平时间和低电平时间还受到 SCL 线上升时间和下降时间的影响，如图 86 中的示意图及方程所示。需注意，SCL 上升和下降时间参数因外部因素不同而变化，例如：

- IO 驱动器特性
- 上拉电阻值
- SCL 线上总电容等

上述特性均超出 DW_apb_i2c 的控制范围。

Figure 86. Impact of SCL Rise Time and Fall Time on Generated SCL



4.3.14.2. Minimum IC_CLK Frequency

This section describes the minimum `ic_clk` frequencies that the DW_apb_i2c supports for each speed mode, and the associated high and low count values. In Slave mode, `IC_SDA_HOLD` (Thd:dat) and `IC_SDA_SETUP` (Tsu:dat) need to be programmed to satisfy the I2C protocol timing requirements. The following examples are for the case where `IC_FS_SPKLEN` is programmed to two.

4.3.14.2.1. Standard Mode (SM), Fast Mode (FM), and Fast Mode Plus (FM+)

This section details how to derive a minimum `ic_clk` value for standard and fast modes of the DW_apb_i2c. Although the following method shows how to do fast mode calculations, you can also use the same method in order to do calculations for standard mode and fast mode plus.

NOTE

The following computations do not consider the SCL_Rise_time and SCL_Fall_time.

Given conditions and calculations for the minimum DW_apb_i2c `ic_clk` value in fast mode:

- Fast mode has data rate of 400kbps; implies `SCL` period of $1/400\text{kHz} = 2.5\mu\text{s}$
- Minimum hcnt value of 14 as a seed value; `IC_HCNT_FS` = 14
- Protocol minimum `SCL` high and low times:
 - `MIN_SCL_LOWtime_FS` = 1300ns
 - `MIN_SCL_HIGHtime_FS` = 600ns

Derived equations:

$$\text{SCL_PERIOD_FS} / (\text{IC_HCNT_FS} + \text{IC_LCNT_FS}) = \text{IC_CLK_PERIOD}$$

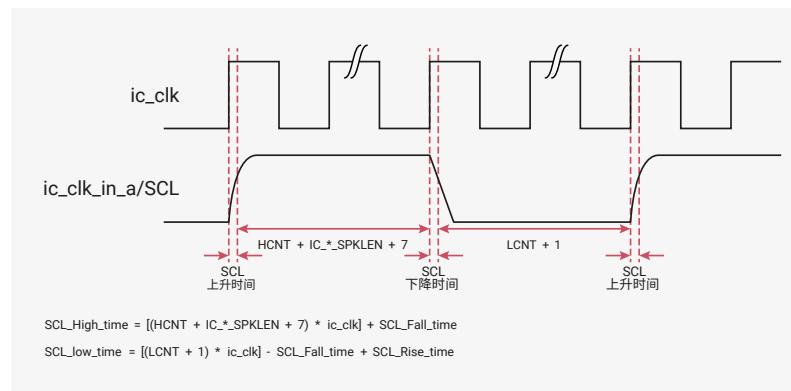
$$\text{IC_LCNT_FS} \times \text{IC_CLK_PERIOD} = \text{MIN_SCL_LOWtime_FS}$$

Combined, the previous equations produce the following:

$$\text{IC_LCNT_FS} \times (\text{SCL_PERIOD_FS} / (\text{IC_LCNT_FS} + \text{IC_HCNT_FS})) = \text{MIN_SCL_LOWtime_FS}$$

Solving for `IC_LCNT_FS`:

图86。 **SCL**上升时间与下降时间对生成信号的影响



4.3.14.2. 最小 **IC_CLK** 频率

本节说明了 DW_apb_i2c 针对各速率模式支持的最小 **ic_clk** 频率及其对应的高低计数值。在从设备模式下，需对 IC_SDA_HOLD (Thd:dat) 和 IC_SDA_SETUP (Tsu:dat) 进行编程，以满足 I2C 协议的时序要求。以下示例适用于 IC_FS_SPKLEN 被设定为 2 的情况。

4.3.14.2.1. 标准模式 (SM)、快速模式 (FM) 及快速模式增强型 (FM+)

本节详细说明如何为 DW_apb_i2c 的标准模式和快速模式导出最小 **ic_clk** 值。尽管以下方法展示了快速模式的计算过程，但同样适用于标准模式及快速加模式的计算。

注意

以下计算未考虑 SCL_Rise_time 和 SCL_Fall_time。

快速模式下 DW_apb_i2c 最小 **ic_clk** 值的条件与计算如下：

- 快速模式的数据传输速率为 400 kbps；即 **SCL** 周期为 $1/400\text{kHz} = 2.5\mu\text{s}$
- 最小初始 hcnt 值为 14； $\text{IC_HCNT_FS} = 14$
- 协议规定的最小 **SCL** 高电平时间与低电平时间如下：
 - $\text{MIN_SCL_LOWtime_FS} = 1300\text{ns}$
 - $\text{MIN_SCL_HIGHtime_FS} = 600\text{ns}$

推导公式：

$$\text{SCL_PERIOD_FS} / (\text{IC_HCNT_FS} + \text{IC_LCNT_FS}) = \text{IC_CLK_PERIOD}$$

$$\text{IC_LCNT_FS} \times \text{IC_CLK_PERIOD} = \text{MIN_SCL_LOWtime_FS}$$

综合前述方程，得到如下结果：

$$\text{IC_LCNT_FS} \times (\text{SCL_PERIOD_FS} / (\text{IC_LCNT_FS} + \text{IC_HCNT_FS})) = \text{MIN_SCL_LOWtime_FS}$$

求解 **IC_LCNT_FS** 如下：

$$\text{IC_LCNT_FS} \times (2.5\mu\text{s} / (\text{IC_LCNT_FS} + 14)) = 1.3\mu\text{s}$$

The previous equation gives:

$$\text{IC_LCNT_FS} = \text{roundup}(15.166) = 16$$

These calculations produce IC_LCNT_FS = 16 and IC_HCNT_FS = 14, giving an `ic_clk` value of:

$$2.5\mu\text{s} / (16 + 14) = 83.3\text{ns} = 12\text{MHz}$$

Testing these results shows that protocol requirements are satisfied.

Table 450 lists the minimum `ic_clk` values for all modes with high and low count values.

Table 450. `ic_clk` in Relation to High and Low Counts

Speed Mode	<code>ic_clkfreq</code> (MHz)	Minimum Value of <code>IC_*_SPKLEN</code>	<code>SCL</code> Low Time in <code>'ic_clk'</code> s	<code>SCL</code> Low Program Value	<code>SCL</code> Low Time	<code>SCL</code> High Time in <code>'ic_clk'</code> s	<code>SCL</code> High Program Value	<code>SCL</code> High Time
SS	2.7	1	13	12	4.7μs	14	6	5.2μs
FS	12.0	1	16	15	1.33μs	14	6	1.16μs
FM+	32	2	16	15	500ns	16	7	500ns

- The IC_*_SCL_LCNT and IC_*_SCL_HCNT registers are programmed using the `SCL` low and high program values in **Table 450**, which are calculated using `SCL` low count minus one, and `SCL` high counts minus eight, respectively. The values in **Table 450** are based on IC_SDA_RX_HOLD = 0. The maximum IC_SDA_RX_HOLD value depends on the `IC_*_CNT` registers in Master mode.
- In order to compute the HCNT and LCNT considering RC timings, use the following equations:
 - $\text{IC_HCNT_*} = [(\text{HCNT} + \text{IC_*_SPKLEN} + 7) * \text{ic_clk}] + \text{SCL_Fall_time}$
 - $\text{IC_LCNT_*} = [(\text{LCNT} + 1) * \text{ic_clk}] - \text{SCL_Fall_time} + \text{SCL_Rise_time}$

4.3.14.3. Calculating High and Low Counts

The calculations below show how to calculate `SCL` high and low counts for each speed mode in the DW_apb_i2c. For the calculations to work, the `ic_clk` frequencies used must not be less than the minimum `ic_clk` frequencies specified in **Table 450**.

The default `ic_clk` period value is set to 100ns, so default `SCL` high and low count values are calculated for each speed mode based on this clock. These values need updating according to the guidelines below.

The equation to calculate the proper number of `ic_clk` signals required for setting the proper `SCL` clocks high and low times is as follows:

$$\text{IC_xCNT} = (\text{ROUNDUP}(\text{MIN_SCL_xxxtime} * \text{OSCFREQ}, 0))$$

`MIN_SCL_HIGHtime` = Minimum High Period
`MIN_SCL_HIGHtime` = 4000ns for 100kbps,
 600ns for 400kbps,
 260ns for 1000kbps,

`MIN_SCL_LOWtime` = Minimum Low Period
`MIN_SCL_LOWtime` = 4700ns for 100kbps,

$$\text{IC_LCNT_FS} \times (2.5\mu\text{s} / (\text{IC_LCNT_FS} + 14)) = 1.3 \mu\text{s}$$

上述方程给出：

$$\text{IC_LCNT_FS} = \text{roundup}(15.166) = 16$$

计算结果为 $\text{IC_LCNT_FS} = 16$, $\text{IC_HCNT_FS} = 14$, 对应的 ic_clk 值为：

$$2.5\mu\text{s} / (16 + 14) = 83.3\text{ns} = 12\text{MHz}$$

经验证，该结果满足协议要求。

表450列出了所有模式下高计数与低计数对应的最小 ic_clk 值。

表450. ic_clk 与高计数与低计数的关系

速度模式	ic_clkfreq (MHz)	IC_SPKLEN 的最小值	SCL 低电平时间, 以 ic_clk 为单位	SCL 低电平程序设定值	SCL 低电平时间	SCL 高电平时间, 单位为 ic_clk	SCL 高电平程序设定值	SCL 高电平时间
SS	2.7	1	13	12	$4.7\mu\text{s}$	14	6	$5.2\mu\text{s}$
FS	12.0	1	16	15	$1.33\mu\text{s}$	14	6	$1.16\mu\text{s}$
FM+	32	2	16	15	500ns	16	7	500ns

- IC_*_SCL_LCNT 和 IC_*_SCL_HCNT 寄存器通过表450中的 SCL 低电平和高电平程序设定值编程, 该值分别由 SCL 低电平时间减一及 SCL 高电平时间减八计算得出。表450中的数值基于 $\text{IC_SDA_RX_HOLD} = 0$ 。最大 IC_SDA_RX_HOLD 值取决于主模式下的 IC_*CNT 寄存器。

- 为计算考虑RC时序的HCNT和LCNT, 使用以下公式:

- $\text{IC_HCNT_*} = [(\text{HCNT} + \text{IC_SPKLEN} + 7) * \text{ic_clk}] + \text{SCL_Fall_time}$
- $\text{IC_LCNT_*} = [(\text{LCNT} + 1) * \text{ic_clk}] - \text{SCL_Fall_time} + \text{SCL_Rise_time}$

4.3.14.3 高电平和低电平计数的计算

以下计算说明如何为 DW_apb_i2c 中各速率模式计算 SCL 高电平和低电平计数。为确保计算有效, 所用 ic_clk 频率不得低于表450中规定的最低 ic_clk 频率。

默认 ic_clk 周期值为 100ns, 基于此时钟计算各速率模式的默认 SCL 高电平和低电平计数值。这些数值需依据以下指南进行更新。

用于计算设置合适 SCL 时钟高、低电平时间所需 ic_clk 信号数量的公式如下：

$$\text{IC_XCNT} = (\text{ROUNDUP}(\text{MIN_SCL_xxxtime} * \text{OSCFREQ}, 0))$$

MIN_SCL_HIGHtime = 最小高电平周期

MIN_SCL_HIGHtime = 100kbps 时为 4000ns, 400
kbps 时为 600ns, 100
0kbps 时为 260ns。

MIN_SCL_LOWtime = 最小低电平周期

MIN_SCL_LOWtime = 100kbps 时为 4700ns,

```

1300ns for 400kbps,
500ns for 1000kbps,
OSCFREQ = ic_clk Clock Frequency (Hz).

```

For example:

```

OSCFREQ = 100MHz
I2Cmode = fast, 400kbps
MIN_SCL_HIGHTime = 600ns.
MIN_SCL_LOWtime = 1300ns.

IC_xCNT = (ROUNDUP(MIN_SCL_HIGH_LOWtime*OSCFREQ,0))

IC_HCNT = (ROUNDUP(600ns * 100MHz,0))
IC_HCNTSCL PERIOD = 60
IC_LCNT = (ROUNDUP(1300ns * 100MHz,0))
IC_LCNTSCL PERIOD = 130
Actual MIN_SCL_HIGHTime = 60*(1/100MHz) = 600ns
Actual MIN_SCL_LOWtime = 130*(1/100MHz) = 1300ns

```

4.3.15. DMA Controller Interface

The DW_apb_i2c has built-in DMA capability; it has a handshaking interface to the DMA Controller to request and control transfers. The APB bus is used to perform the data transfer to or from the DMA. DMA transfers are transferred as single accesses as data rate is relatively low.

4.3.15.1. Enabling the DMA Controller Interface

To enable the DMA Controller interface on the DW_apb_i2c, you must write the DMA Control Register ([IC_DMA_CR](#)). Writing a one into the TDMAE bit field of [IC_DMA_CR](#) register enables the DW_apb_i2c transmit handshaking interface. Writing a one into the RDMAE bit field of the [IC_DMA_CR](#) register enables the DW_apb_i2c receive handshaking interface.

4.3.15.2. Overview of Operation

The DMA Controller is programmed with the number of data items (transfer count) that are to be transmitted or received by DW_apb_i2c.

The transfer is broken into single transfers on the bus, each initiated by a request from the DW_apb_i2c.

For example, where the transfer count programmed into the DMA Controller is four. The DMA transfer consists of a series of four single transactions. If the DW_apb_i2c makes a transmit request to this channel, a single data item is written to the DW_apb_i2c TX FIFO. Similarly, if the DW_apb_i2c makes a receive request to this channel, a single data item is read from the DW_apb_i2c RX FIFO. Four separate requests must be made to this DMA channel before all four data items are written or read.

4.3.15.3. Watermark Levels

In DW_apb_i2c the registers for setting watermarks to allow DMA bursts do not need to be set to anything other than their reset value. Specifically [IC_DMA_TDLR](#) and [IC_DMA_RDLR](#) can be left at reset values of zero. This is because only single transfers are needed due to the low bandwidth of I2C relative to system bandwidth, and also the DMA controller

400kbps时为1300ns,
1000kbps时为500ns。

$\text{OSCFREQ} = \text{ic_clk}$ 时钟频率 (Hz)

例如：

```

OSCFREQ = 100MHz
I2Cmode = 快速模式, 400kbps
MIN_SCL_HIGHtime = 600ns。
MIN_SCL_LOWtime = 1300ns。

IC_xCNT = (ROUNDUP(MIN_SCL_HIGH_LOWtime*OSCFREQ, 0))

IC_HCNT = (ROUNDUP(600ns * 100MHz, 0))
IC_HCNTSCL 周期 = 60
IC_LCNT = (ROUNDUP(1300ns * 100MHz, 0))
IC_LCNTSCL 周期 = 130
实际最小 SCL 高电平时间 = 60 × (1/100MHz) = 600ns
实际最小 SCL 低电平时间 = 130 × (1/100MHz) = 1300ns

```

4.3.15. DMA 控制器接口

DW_apb_i2c 具备内置 DMA 功能；其通过握手接口与 DMA 控制器通信，以请求并控制传输。APB 总线用于在 DMA 之间执行数据传输。由于数据速率较低，DMA 传输以单次访问的形式进行。

4.3.15.1. 启用 DMA 控制器接口

要启用 DW_apb_i2c 上的 DMA 控制器接口，必须写入 DMA 控制寄存器 (IC_DMA_CR)。
向 IC_DMA_CR 寄存器的 TDMAE 位域写入 1 以启用 DW_apb_i2c 的发送握手机制。
向 IC_DMA_CR 寄存器的 RDMAE 位域写入 1 以启用 DW_apb_i2c 的接收握手机制。

4.3.15.2. 操作概述

DMA 控制器被编程为传输或接收 DW_apb_i2c 指定数量的数据项（传输计数）。

传输被拆分为总线上的单个传输，每次由 DW_apb_i2c 发起请求。

例如，若 DMA 控制器中设置的传输计数为四。DMA 传输包含连续的四次单独事务。若 DW_apb_i2c 向该通道发起发送请求，则会向 DW_apb_i2c 的 TX FIFO 写入一个数据项。同理，若 DW_apb_i2c 向该通道发起接收请求，则会从 DW_apb_i2c 的 RX FIFO 读取一个数据项。完成所有四个数据项的读写之前，必须对该 DMA 通道发起四次独立请求。

4.3.15.3. 水印级别

在 DW_apb_i2c 中，用于设置水印以允许 DMA 突发的寄存器无需设置为除复位值外的其他值。具体而言，IC_DMA_TDLR 和 IC_DMA_RDLR 可保持为复位值零。这是因为相较于系统带宽，I2C 的带宽较低，仅需单次传输，同时 DMA 控制器

normally has highest priority on the system bus so will generally complete very quickly.

4.3.16. Operation of Interrupt Registers

Table 451 lists the operation of the DW_apb_i2c interrupt registers and how they are set and cleared. Some bits are set by hardware and cleared by software, whereas other bits are set and cleared by hardware.

Table 451. Clearing and Setting of Interrupt Registers

Interrupt Bit Fields	Set by Hardware/Cleared by Software	Set and Cleared by Hardware
RESTART_DET	Y	N
GEN_CALL	Y	N
START_DET	Y	N
STOP_DET	Y	N
ACTIVITY	Y	N
RX_DONE	Y	N
TX_ABRT	Y	N
RD_REQ	Y	N
TX_EMPTY	N	Y
TX_OVER	Y	N
RX_FULL	N	Y
RX_OVER	Y	N
RX_UNDER	Y	N

4.3.17. List of Registers

The I2C0 and I2C1 registers start at base addresses of `0x40044000` and `0x40048000` respectively (defined as `I2C0_BASE` and `I2C1_BASE` in SDK).

i NOTE

You may see references to configuration constants in the I2C register descriptions; these are **fixed** values, set at hardware design time. A full list of their values can be found in https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_regs/include/hardware/regs/i2c.h

Table 452. List of I2C registers

Offset	Name	Info
0x00	<code>IC_CON</code>	I2C Control Register
0x04	<code>IC_TAR</code>	I2C Target Address Register
0x08	<code>IC_SAR</code>	I2C Slave Address Register
0x10	<code>IC_DATA_CMD</code>	I2C Rx/Tx Data Buffer and Command Register
0x14	<code>IC_SS_SCL_HCNT</code>	Standard Speed I2C Clock SCL High Count Register
0x18	<code>IC_SS_SCL_LCNT</code>	Standard Speed I2C Clock SCL Low Count Register
0x1c	<code>IC_FS_SCL_HCNT</code>	Fast Mode or Fast Mode Plus I2C Clock SCL High Count Register
0x20	<code>IC_FS_SCL_LCNT</code>	Fast Mode or Fast Mode Plus I2C Clock SCL Low Count Register

通常在系统总线上优先级最高，因此一般能非常快速地完成传输。

4.3.16. 中断寄存器操作

表451列出了 DW_apb_i2c 中断寄存器的操作方式及其设置和清除的方法。某些位由硬件设置并由软件清除，而其他位则由硬件设置和清除。

表451。中断寄存器的清除与设置

中断位字段	硬件设置／软件清除	硬件设置与清除
RESTART_DET	是	否
GEN_CALL	是	否
START_DET	是	否
STOP_DET	是	否
活动	是	否
RX_DONE	是	否
TX_ABRT	是	否
RD_REQ	是	否
TX_EMPTY	否	是
TX_OVER	是	否
RX_FULL	否	是
RX_OVER	是	否
RX_UNDER	是	否

4.3.17. 寄存器列表

I2C0 和 I2C1 寄存器的基址分别为 `0x40044000` 和 `0x40048000`（在 SDK 中定义为 I2C0_BASE 和 I2C1_BASE）。

① 注意

您可能会在I2C寄存器描述中看到配置常量的引用；这些值是固定的，于硬件设计阶段设定。完整的取值列表可见于 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_regs/include/hardware/regs/i2c.h

表452. I2C 寄存器列表

偏移量	名称	说明
0x00	IC_CON	I2C控制寄存器
0x04	IC_TAR	I2C目标地址寄存器
0x08	IC_SAR	I2C从属地址寄存器
0x10	IC_DATA_CMD	I2C接收/发送数据缓冲及命令寄存器
0x14	IC_SS_SCL_HCNT	标准速度I2C时钟SCL高电平计数寄存器
0x18	IC_SS_SCL_LCNT	标准速度I2C时钟SCL低电平计数寄存器
0x1c	IC_FS_SCL_HCNT	快速模式或增强快速模式I2C时钟SCL高电平计数寄存器
0x20	IC_FS_SCL_LCNT	快速模式或增强快速模式I2C时钟SCL低电平计数寄存器

Offset	Name	Info
0x2c	IC_INTR_STAT	I2C Interrupt Status Register
0x30	IC_INTR_MASK	I2C Interrupt Mask Register
0x34	IC_RAW_INTR_STAT	I2C Raw Interrupt Status Register
0x38	IC_RX_TL	I2C Receive FIFO Threshold Register
0x3c	IC_TX_TL	I2C Transmit FIFO Threshold Register
0x40	IC_CLR_INTR	Clear Combined and Individual Interrupt Register
0x44	IC_CLR_RX_UNDER	Clear RX_UNDER Interrupt Register
0x48	IC_CLR_RX_OVER	Clear RX_OVER Interrupt Register
0x4c	IC_CLR_TX_OVER	Clear TX_OVER Interrupt Register
0x50	IC_CLR_RD_REQ	Clear RD_REQ Interrupt Register
0x54	IC_CLR_TX_ABRT	Clear TX_ABRT Interrupt Register
0x58	IC_CLR_RX_DONE	Clear RX_DONE Interrupt Register
0x5c	IC_CLR_ACTIVITY	Clear ACTIVITY Interrupt Register
0x60	IC_CLR_STOP_DET	Clear STOP_DET Interrupt Register
0x64	IC_CLR_START_DET	Clear START_DET Interrupt Register
0x68	IC_CLR_GEN_CALL	Clear GEN_CALL Interrupt Register
0x6c	IC_ENABLE	I2C ENABLE Register
0x70	IC_STATUS	I2C STATUS Register
0x74	IC_TXFLR	I2C Transmit FIFO Level Register
0x78	IC_RXFLR	I2C Receive FIFO Level Register
0x7c	IC_SDA_HOLD	I2C SDA Hold Time Length Register
0x80	IC_TX_ABRT_SOURCE	I2C Transmit Abort Source Register
0x84	IC_SLV_DATA_NACK_ONLY	Generate Slave Data NACK Register
0x88	IC_DMA_CR	DMA Control Register
0x8c	IC_DMA_TDLR	DMA Transmit Data Level Register
0x90	IC_DMA_RDLR	DMA Transmit Data Level Register
0x94	IC_SDA_SETUP	I2C SDA Setup Register
0x98	IC_ACK_GENERAL_CALL	I2C ACK General Call Register
0x9c	IC_ENABLE_STATUS	I2C Enable Status Register
0xa0	IC_FS_SPKLEN	I2C SS, FS or FM+ spike suppression limit
0xa8	IC_CLR_RESTART_DET	Clear RESTART_DET Interrupt Register
0xf4	IC_COMP_PARAM_1	Component Parameter Register 1
0xf8	IC_COMP_VERSION	I2C Component Version Register
0xfc	IC_COMP_TYPE	I2C Component Type Register

I2C: IC_CON Register

偏移量	名称	说明
0x2c	IC_INTR_STAT	I2C中断状态寄存器
0x30	IC_INTR_MASK	I2C中断屏蔽寄存器
0x34	IC_RAW_INTR_STAT	I2C原始中断状态寄存器
0x38	IC_RX_TL	I2C接收FIFO阈值寄存器
0x3c	IC_TX_TL	I2C发送FIFO阈值寄存器
0x40	IC_CLR_INTR	清除组合中断及单个中断寄存器
0x44	IC_CLR_RX_UNDER	清除 RX_UNDER 中断寄存器
0x48	IC_CLR_RX_OVER	清除 RX_OVER 中断寄存器
0x4c	IC_CLR_TX_OVER	清除 TX_OVER 中断寄存器
0x50	IC_CLR_RD_REQ	清除 RD_REQ 中断寄存器
0x54	IC_CLR_TX_ABRT	清除 TX_ABRT 中断寄存器
0x58	IC_CLR_RX_DONE	清除 RX_DONE 中断寄存器
0x5c	IC_CLR_ACTIVITY	清除 ACTIVITY 中断寄存器
0x60	IC_CLR_STOP_DET	清除 STOP_DET 中断寄存器
0x64	IC_CLR_START_DET	清除 START_DET 中断寄存器
0x68	IC_CLR_GEN_CALL	清除 GEN_CALL 中断寄存器
0x6c	IC_ENABLE	I2C 使能寄存器
0x70	IC_STATUS	I2C 状态寄存器
0x74	IC_TXFLR	I2C 发送 FIFO 水平寄存器
0x78	IC_RXFLR	I2C 接收 FIFO 水平寄存器
0x7c	IC_SDA_HOLD	I2C SDA 保持时间寄存器
0x80	IC_TX_ABRT_SOURCE	I2C 发送中止源寄存器
0x84	IC_SLV_DATA_NACK_ONLY	生成从设备数据 NACK 寄存器
0x88	IC_DMA_CR	DMA 控制寄存器
0x8c	IC_DMA_TDLR	DMA 传输数据级别寄存器
0x90	IC_DMA_RDLR	DMA 传输数据级别寄存器
0x94	IC_SDA_SETUP	I2C SDA 设置寄存器
0x98	IC_ACK_GENERAL_CALL	I2C 应答通用调用寄存器
0x9c	IC_ENABLE_STATUS	I2C 使能状态寄存器
0xa0	IC_FS_SPKLEN	I2C SS、FS 或 FM+ 峰值抑制限制
0xa8	IC_CLR_RESTART_DET	清除 RESTART_DET 中断寄存器
0xf4	IC_COMP_PARAM_1	组件参数寄存器 1
0xf8	IC_COMP_VERSION	I2C 组件版本寄存器
0xfc	IC_COMP_TYPE	I2C 组件类型寄存器

I2C: IC_CON 寄存器

Offset: 0x00

Description

I2C Control Register. This register can be written only when the DW_apb_i2c is disabled, which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect.

Read/Write Access: - bit 10 is read only. - bit 11 is read only - bit 16 is read only - bit 17 is read only - bits 18 and 19 are read only.

Table 453. IC_CON Register

Bits	Description	Type	Reset
31:11	Reserved.	-	-
10	STOP_DET_IF_MASTER_ACTIVE: Master issues the STOP_DET interrupt irrespective of whether master is active or not	RO	0x0
9	RX_FIFO_FULL_HLD_CTRL: This bit controls whether DW_apb_i2c should hold the bus when the Rx FIFO is physically full to its RX_BUFFER_DEPTH, as described in the IC_RX_FULL_HLD_BUS_EN parameter. Reset value: 0x0.	RW	0x0
	Enumerated values:		
	0x0 → DISABLED: Overflow when RX_FIFO is full		
	0x1 → ENABLED: Hold bus when RX_FIFO is full		
8	TX_EMPTY_CTRL: This bit controls the generation of the TX_EMPTY interrupt, as described in the IC_RAW_INTR_STAT register. Reset value: 0x0.	RW	0x0
	Enumerated values:		
	0x0 → DISABLED: Default behaviour of TX_EMPTY interrupt		
	0x1 → ENABLED: Controlled generation of TX_EMPTY interrupt		
7	STOP_DET_IFADDRESSED: In slave mode: - 1'b1: issues the STOP_DET interrupt only when it is addressed. - 1'b0: issues the STOP_DET irrespective of whether it's addressed or not. Reset value: 0x0 NOTE: During a general call address, this slave does not issue the STOP_DET interrupt if STOP_DET_IF_ADDRESSED = 1'b1, even if the slave responds to the general call address by generating ACK. The STOP_DET interrupt is generated only when the transmitted address matches the slave address (SAR). Enumerated values:	RW	0x0
	0x0 → DISABLED: slave issues STOP_DET intr always		
	0x1 → ENABLED: slave issues STOP_DET intr only if addressed		
6	IC_SLAVE_DISABLE: This bit controls whether I2C has its slave disabled, which means once the presetn signal is applied, then this bit is set and the slave is disabled. If this bit is set (slave is disabled), DW_apb_i2c functions only as a master and does not perform any action that requires a slave. NOTE: Software should ensure that if this bit is written with 0, then bit 0 should also be written with a 0.	RW	0x1

偏移: 0x00

描述

I2C 控制寄存器。该寄存器仅在 DW_apb_i2c 被禁用时可写，对应 IC_ENABLE[0] 寄存器设为 0。其他时间的写入操作无效。

读/写权限：- 位 10 为只读。- 位 11 为只读 - 位 16 为只读 - 位 17 为只读 - 位 18 和位 19 为只读。

表 453. IC_CON
寄存器

位	描述	类型	复位值
31:11	保留。	-	-
10	STOP_DET_IF_MASTER_ACTIVE : 无论主设备是否处于激活状态，主机均会触发 STOP_DET 中断	只读	0x0
9	RX_FIFO_FULL_HLD_CTRL : 该位控制当接收 FIFO 物理填满至 RX_BUFFER_D_EPTH 时，DW_apb_i2c 是否应阻止总线访问，如 IC_RX_FULL_HLD_BU_S_EN 参数中所述。 复位值: 0x0。	读写	0x0
	枚举值:		
	0x0 → 禁用: 当 RX_FIFO 满时发生溢出		
	0x1 → 启用: 当 RX_FIFO 满时保持总线		
8	TX_EMPTY_CTRL : 该位控制 TX_EMPTY 中断的生成，如 IC_RAW_INTR_STA_T 寄存器所述。 复位值: 0x0。	读写	0x0
	枚举值:		
	0x0 → 禁用: TX_EMPTY 中断的默认行为		
	0x1 → 启用: 控制 TX_EMPTY 中断的生成		
7	STOP_DET_IFADDRESSED : 在从属模式下: - 1'b1: 仅当被寻址时发出 STOP_DET 中断。- 1'b0: 无论是否被寻址，均发出 STOP_DET 中断。复位值: 0x0 注意: 在通用调用地址期间，如 STOP_DET_IF_ADDRESSED=1'b1，即使从属通过生成 ACK 对通用调用地址作出响应，亦不会发出 STOP_DET 中断。STOP_DET 中断仅在所传输地址与从属地址 (SAR) 匹配时生成。	读写	0x0
	枚举值:		
	0x0 → 已禁用: 从机始终发出 STOP_DET 中断		
	0x1 → 已启用: 从机仅在被寻址时发出 STOP_DET 中断		
6	IC_SLAVE_DISABLE : 此位控制 I2C 从机是否被禁用，即一旦施加预设信号，该位被置位，从机即被禁用。 如果该位被置位（从机被禁用），则 DW_apb_i2c 仅作为主机工作，不执行任何需要从机的操作。 注意: 软件应确保如果该位写入 0，则位0也应写入 0。	读写	0x1

Bits	Description	Type	Reset
	Enumerated values:		
	0x0 → SLAVE_ENABLED: Slave mode is enabled		
	0x1 → SLAVE_DISABLED: Slave mode is disabled		
5	IC_RESTART_EN : Determines whether RESTART conditions may be sent when acting as a master. Some older slaves do not support handling RESTART conditions; however, RESTART conditions are used in several DW_apb_i2c operations. When RESTART is disabled, the master is prohibited from performing the following functions: - Sending a START BYTE - Performing any high-speed mode operation - High-speed mode operation - Performing direction changes in combined format mode - Performing a read operation with a 10-bit address By replacing RESTART condition followed by a STOP and a subsequent START condition, split operations are broken down into multiple DW_apb_i2c transfers. If the above operations are performed, it will result in setting bit 6 (TX_ABRT) of the IC_RAW_INTR_STAT register. Reset value: ENABLED	RW	0x1
	Enumerated values:		
	0x0 → DISABLED: Master restart disabled		
	0x1 → ENABLED: Master restart enabled		
4	IC_10BITADDR_MASTER : Controls whether the DW_apb_i2c starts its transfers in 7- or 10-bit addressing mode when acting as a master. - 0: 7-bit addressing - 1: 10-bit addressing	RW	0x0
	Enumerated values:		
	0x0 → ADDR_7BITS: Master 7Bit addressing mode		
	0x1 → ADDR_10BITS: Master 10Bit addressing mode		
3	IC_10BITADDR_SLAVE : When acting as a slave, this bit controls whether the DW_apb_i2c responds to 7- or 10-bit addresses. - 0: 7-bit addressing. The DW_apb_i2c ignores transactions that involve 10-bit addressing; for 7-bit addressing, only the lower 7 bits of the IC_SAR register are compared. - 1: 10-bit addressing. The DW_apb_i2c responds to only 10-bit addressing transfers that match the full 10 bits of the IC_SAR register.	RW	0x0
	Enumerated values:		
	0x0 → ADDR_7BITS: Slave 7Bit addressing		
	0x1 → ADDR_10BITS: Slave 10Bit addressing		

位	描述	类型	复位值
	枚举值：		
	0x0 → 从机已启用：从机模式被启用		
	0x1 → 从机已禁用：从机模式被禁用		
5	IC_RESTART_EN : 决定作为主机时是否允许发送 RESTART 条件。部分旧版从机不支持处理 RESTART 条件；然而，在多个 DW_apb_i2c 操作中会使用 RESTART 条件。当禁用 RESTART 时，主设备禁止执行以下操作：- 发送 START 字节 - 执行任何高速模式操作 - 在组合格式模式下进行方向切换 - 使用 10 位地址执行读取操作。通过用 RESTART 条件替代随后紧跟的 STOP 以及再次的 START 条件，分割操作被拆分为多个 DW_apb_i2c 传输。若执行上述操作，将导致设置 IC_RAW_INTR_STAT 寄存器第 6 位 (TX_AB RT)。	读写	0x1
	复位值：ENABLED		
	枚举值：		
	0x0 → DISABLED：主设备重启功能被禁用		
	0x1 → ENABLED：主设备重启功能被启用		
4	IC_10BITADDR_MASTER : 控制 DW_apb_i2c 作为主设备时，传输启动所采用的 7 位或 10 位寻址模式。- 0: 7 位寻址 - 1: 10 位寻址	读写	0x0
	枚举值：		
	0x0 → ADDR_7BITS：主设备 7 位寻址模式		
	0x1 → ADDR_10BITS：主机10位寻址模式		
3	IC_10BITADDR_SLAVE : 作为从机时，该位控制DW_apb_i2c响应7位还是10位地址。- 0: 7位寻址。DW_apb_i2c会忽略涉及10位寻址的事务；对于7位寻址，仅比较IC_SAR寄存器的低7位。- 1: 10位寻址。DW_apb_i2c仅响应与IC_SAR寄存器完整10位匹配的10位寻址传输。	读写	0x0
	枚举值：		
	0x0 → ADDR_7BITS：从机7位寻址		
	0x1 → ADDR_10BITS：从机10位寻址		

Bits	Description	Type	Reset
2:1	<p>SPEED: These bits control at which speed the DW_apb_i2c operates; its setting is relevant only if one is operating the DW_apb_i2c in master mode. Hardware protects against illegal values being programmed by software. These bits must be programmed appropriately for slave mode also, as it is used to capture correct value of spike filter as per the speed mode.</p> <p>This register should be programmed only with a value in the range of 1 to IC_MAX_SPEED_MODE; otherwise, hardware updates this register with the value of IC_MAX_SPEED_MODE.</p> <p>1: standard mode (100 kbit/s) 2: fast mode (<=400 kbit/s) or fast mode plus (<=1000Kbit/s) 3: high speed mode (3.4 Mbit/s)</p> <p>Note: This field is not applicable when IC_ULTRA_FAST_MODE=1</p>	RW	0x2
	Enumerated values:		
	0x1 → STANDARD: Standard Speed mode of operation		
	0x2 → FAST: Fast or Fast Plus mode of operation		
	0x3 → HIGH: High Speed mode of operation		
0	<p>MASTER_MODE: This bit controls whether the DW_apb_i2c master is enabled.</p> <p>NOTE: Software should ensure that if this bit is written with '1' then bit 6 should also be written with a '1'.</p>	RW	0x1
	Enumerated values:		
	0x0 → DISABLED: Master mode is disabled		
	0x1 → ENABLED: Master mode is enabled		

I2C: IC_TAR Register

Offset: 0x04

Description

I2C Target Address Register

This register is 12 bits wide, and bits 31:12 are reserved. This register can be written to only when IC_ENABLE[0] is set to 0.

Note: If the software or application is aware that the DW_apb_i2c is not using the TAR address for the pending commands in the Tx FIFO, then it is possible to update the TAR address even while the Tx FIFO has entries (IC_STATUS[2]= 0). - It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I2C slave only.

Table 454. IC_TAR Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	<p>SPECIAL: This bit indicates whether software performs a Device-ID or General Call or START BYTE command. - 0: ignore bit 10 GC_OR_START and use IC_TAR normally - 1: perform special I2C command as specified in Device_ID or GC_OR_START bit Reset value: 0x0</p>	RW	0x0

位	描述	类型	复位值
2:1	<p>SPEED: 这些位用于控制DW_apb_i2c的工作速度；该设置仅在以主机模式操作DW_apb_i2c时适用。 硬件防止非法值被软件编入。 这些位也必须为从模式适当编程，因为它们用于根据速度模式捕获峰值滤波器的正确值。</p> <p>此寄存器应仅编程为1至IC_MAX_SPEED_MODE范围内的值；否则，硬件会将此寄存器更新为IC_MAX_SPEED_MODE的值。</p> <p>1: 标准模式 (100 kbit/s) 2: 快速模式 (<=400 kbit/s) 或快速增强模式 (<=1000 kbit/s) 3: 高速模式 (3.4 Mbit/s)</p> <p>注：当IC_ULTRA_FAST_MODE=1时，此字段不适用。</p>	读写	0x2
	枚举值：		
	0x1 → STANDARD：标准速度操作模式		
	0x2 → FAST：快速或快速增强操作模式		
	0x3 → HIGH：高速操作模式		
0	<p>MASTER_MODE: 该位控制是否启用DW_apb_i2c主控。</p> <p>注意：软件应确保若此位写入‘1’，则第6位亦应写入‘1’。</p>	读写	0x1
	枚举值：		
	0x0 → 禁用：主模式被禁用		
	0x1 → 启用：主模式被启用		

I2C: IC_TAR寄存器

偏移量: 0x04

说明

I2C目标地址寄存器

该寄存器宽度为12位，第31至12位为保留位。仅当IC_ENABLE[0]设置为0时，方可写入该寄存器。

注意：若软件或应用程序确认DW_apb_i2c未将TAR地址用于Tx FIFO中的待处理命令，则即使Tx FIFO有条目 (IC_STATUS[2]=0)，仍可更新TAR地址。- 若DW_apb_i2c仅作为I2C从设备启用，则无需对该寄存器进行写入操作。

表454. IC_TAR
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	<p>SPECIAL: 该位指示软件是否执行设备ID、通用呼叫或起始字节命令。- 0: 忽略第10位GC_OR_START，正常使用IC_TAR-1: 执行Device_ID或GC_OR_START位指定的特殊I2C命令 复位值: 0x0</p>	读写	0x0

Bits	Description	Type	Reset
	Enumerated values:		
	0x0 → DISABLED: Disables programming of GENERAL_CALL or START_BYT transmission		
	0x1 → ENABLED: Enables programming of GENERAL_CALL or START_BYT transmission		
10	GC_OR_START : If bit 11 (SPECIAL) is set to 1 and bit 13(Device-ID) is set to 0, then this bit indicates whether a General Call or START byte command is to be performed by the DW_apb_i2c. - 0: General Call Address - after issuing a General Call, only writes may be performed. Attempting to issue a read command results in setting bit 6 (TX_ABRT) of the IC_RAW_INTR_STAT register. The DW_apb_i2c remains in General Call mode until the SPECIAL bit value (bit 11) is cleared. - 1: START BYTE Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → GENERAL_CALL: GENERAL_CALL byte transmission		
	0x1 → START_BYT: START byte transmission		
9:0	IC_TAR : This is the target address for any master transaction. When transmitting a General Call, these bits are ignored. To generate a START BYTE, the CPU needs to write only once into these bits. If the IC_TAR and IC_SAR are the same, loopback exists but the FIFOs are shared between master and slave, so full loopback is not feasible. Only one direction loopback mode is supported (simplex), not duplex. A master cannot transmit to itself; it can transmit to only a slave.	RW	0x055

I2C: IC_SAR Register

Offset: 0x08

Description

I2C Slave Address Register

Table 455. IC_SAR Register

Bits	Description	Type	Reset
31:10	Reserved.	-	-
9:0	IC_SAR : The IC_SAR holds the slave address when the I2C is operating as a slave. For 7-bit addressing, only IC_SAR[6:0] is used. This register can be written only when the I2C interface is disabled, which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect. Note: The default values cannot be any of the reserved address locations: that is, 0x00 to 0x07, or 0x78 to 0x7f. The correct operation of the device is not guaranteed if you program the IC_SAR or IC_TAR to a reserved value. Refer to Table 448 for a complete list of these reserved values.	RW	0x055

I2C: IC_DATA_CMD Register

Offset: 0x10

位	描述	类型	复位值
	枚举值：		
	0x0 → 禁用：禁止编程GENERAL_CALL或START_BYT传输		
	0x1 → 启用：允许编程GENERAL_CALL或START_BYT传输		
10	GC_OR_START : 当第11位 (SPECIAL) 为1且第13位 (Device-ID) 为0时，此位指示DW_apb_i2c是否执行General Call或START字节命令。- 0: General Call 地址 - 发出General Call后，仅允许写操作。尝试发出读取命令时，将设置IC_RAW_INTR_STAT寄存器的第6位 (TX_ABRT)。DW_apb_i2c将保持在General Call模式，直至SPECIAL位 (第11位) 被清除。- 1: START BYTE 重置值: 0x0	读写	0x0
	枚举值：		
	0x0 → GENERAL_CALL：GENERAL_CALL 字节传输		
	0x1 → START_BYT：START 字节传输		
9:0	IC_TAR : 此地址为任何主设备事务的目标地址。发送 General Call 时，这些位将被忽略。生成 START BYT 时，CPU 只需写入这些位一次即可。 若 IC_TAR 与 IC_SAR 相同，则存在回环，但主从共享 FIFO，故不支持完全回环。仅支持单向回环模式（单工），不支持双工模式。主设备不可向自身发送数据；只能向从设备发送数据。	读写	0x055

I2C: IC_SAR 寄存器

偏移: 0x08

描述

I2C从属地址寄存器

表 455. IC_SAR 寄存器

位	描述	类型	复位值
31:10	保留。	-	-
9:0	IC_SAR : I2C 作为从设备时，IC_SAR 保存从设备地址。对于7位寻址，仅使用IC_SAR[6:0]。 仅当I2C接口被禁用（对应IC_ENABLE[0]寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。 注意：默认值不得为任何保留地址区间内的地址，即0x00至0x07，或0x78至0x7F。若将IC_SAR或IC_TAR设置为保留值，设备的正确运行将无法保证。 有关完整保留值列表，请参见表448。	读写	0x055

I2C: IC_DATA_CMD寄存器

偏移: 0x10

Description

I2C Rx/Tx Data Buffer and Command Register; this is the register the CPU writes to when filling the TX FIFO and the CPU reads from when retrieving bytes from RX FIFO.

The size of the register changes as follows:

Write: - 11 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=1 - 9 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=0 Read: - 12 bits when IC_FIRST_DATA_BYTE_STATUS = 1 - 8 bits when IC_FIRST_DATA_BYTE_STATUS = 0 Note: In order for the DW_apb_i2c to continue acknowledging reads, a read command should be written for every byte that is to be received; otherwise the DW_apb_i2c will stop acknowledging.

*Table 456.
IC_DATA_CMD
Register*

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11	FIRST_DATA_BYTE : Indicates the first data byte received after the address phase for receive transfer in Master receiver or Slave receiver mode. Reset value : 0x0 NOTE: In case of APB_DATA_WIDTH=8, 1. The user has to perform two APB Reads to IC_DATA_CMD in order to get status on 11 bit. 2. In order to read the 11 bit, the user has to perform the first data byte read [7:0] (offset 0x10) and then perform the second read [15:8] (offset 0x11) in order to know the status of 11 bit (whether the data received in previous read is a first data byte or not). 3. The 11th bit is an optional read field, user can ignore 2nd byte read [15:8] (offset 0x11) if not interested in FIRST_DATA_BYTE status.	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: Sequential data byte received		
	0x1 → ACTIVE: Non sequential data byte received		
10	RESTART : This bit controls whether a RESTART is issued before the byte is sent or received. 1 - If IC_RESTART_EN is 1, a RESTART is issued before the data is sent/received (according to the value of CMD), regardless of whether or not the transfer direction is changing from the previous command; if IC_RESTART_EN is 0, a STOP followed by a START is issued instead. 0 - If IC_RESTART_EN is 1, a RESTART is issued only if the transfer direction is changing from the previous command; if IC_RESTART_EN is 0, a STOP followed by a START is issued instead. Reset value: 0x0	SC	0x0
	Enumerated values:		
	0x0 → DISABLE: Don't Issue RESTART before this command		
	0x1 → ENABLE: Issue RESTART before this command		

描述

I2C接收/发送数据缓冲及命令寄存器；CPU向TX FIFO写入数据及从RX FIFO读取字节时，均通过此寄存器进行操作。

寄存器位宽变化如下：

写入： - 当IC_EMPTYFIFO_HOLD_MASTER_EN=1时为11位 - 当IC_EMPTYFIFO_HOLD_MASTER_EN=0时为9位
 读取： - 当IC_FIRST_DATA_BYTE_STATUS=1时为12位 - 当IC_FIRST_DATA_BYTE_STATUS=0时为8位
 注：为确保DW_apb_i2c持续响应读取，每个待接收字节应对应写入一个读取命令；
 否则 DW_apb_i2c 将停止响应确认。

表 456。
IC_DATA_CMD
 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11	FIRST_DATA_BYTE : 表示在主设备接收或从设备接收模式下，地址阶段后接收到的第一个数据字节。 复位值：0x0 注意：当 APB_DATA_WIDTH=8 时， 1. 用户必须对 IC_DATA_CMD 执行两次 APB 读取，以获取第 11 位的状态。 2. 为读取第 11 位，用户需首先读取第一个数据字节 [7:0]（偏移地址 0x10），随后执行第二次读取 [15:8]（偏移地址 0x11），以确定第 11 位状态（即上次读取的数据是否为第一个数据字节）。 3. 第 11 位为可选读取字段，用户可忽略第二次读取的 [15:8]（偏移地址 0x11），若不关心 FIRST_DATA_BYTE 状态。	只读	0x0
	枚举值：		
	0x0 → 非活动：接收到顺序数据字节		
	0x1 → 激活：接收到非顺序数据字节		
10	RESTART : 该位控制是否在数据字节发送或接收之前发出 RESTART。 1 - 若 IC_RESTART_EN 为 1，则无论传输方向是否改变，均会在数据发送/接收之前（依据 CMD 值）发出 RESTART；若 IC_RESTART_EN 为 0，则改为先发出 STOP 再发出 START。 0 - 若 IC_RESTART_EN 为 1，仅当传输方向与前一命令不同，才发出 RESTART；若 IC_RESTART_EN 为 0，则改为先发出 STOP 再发出 START。 复位值：0x0	SC	0x0
	枚举值：		
	0x0 → 禁用：此命令前不发出 RESTART		
	0x1 → 启用：此命令前发出 RESTART		

Bits	Description	Type	Reset
9	<p>STOP: This bit controls whether a STOP is issued after the byte is sent or received.</p> <p>- 1 - STOP is issued after this byte, regardless of whether or not the Tx FIFO is empty. If the Tx FIFO is not empty, the master immediately tries to start a new transfer by issuing a START and arbitrating for the bus. - 0 - STOP is not issued after this byte, regardless of whether or not the Tx FIFO is empty. If the Tx FIFO is not empty, the master continues the current transfer by sending/receiving data bytes according to the value of the CMD bit. If the Tx FIFO is empty, the master holds the SCL line low and stalls the bus until a new command is available in the Tx FIFO. Reset value: 0x0</p>	SC	0x0
	Enumerated values:		
	0x0 → DISABLE: Don't Issue STOP after this command		
	0x1 → ENABLE: Issue STOP after this command		
8	<p>CMD: This bit controls whether a read or a write is performed. This bit does not control the direction when the DW_apb_i2con acts as a slave. It controls only the direction when it acts as a master.</p> <p>When a command is entered in the TX FIFO, this bit distinguishes the write and read commands. In slave-receiver mode, this bit is a 'don't care' because writes to this register are not required. In slave-transmitter mode, a '0' indicates that the data in IC_DATA_CMD is to be transmitted.</p> <p>When programming this bit, you should remember the following: attempting to perform a read operation after a General Call command has been sent results in a TX_ABRT interrupt (bit 6 of the IC_RAW_INTR_STAT register), unless bit 11 (SPECIAL) in the IC_TAR register has been cleared. If a '1' is written to this bit after receiving a RD_REQ interrupt, then a TX_ABRT interrupt occurs.</p> <p>Reset value: 0x0</p>	SC	0x0
	Enumerated values:		
	0x0 → WRITE: Master Write Command		
	0x1 → READ: Master Read Command		
7:0	<p>DAT: This register contains the data to be transmitted or received on the I2C bus. If you are writing to this register and want to perform a read, bits 7:0 (DAT) are ignored by the DW_apb_i2c. However, when you read this register, these bits return the value of data received on the DW_apb_i2c interface.</p> <p>Reset value: 0x0</p>	RW	0x00

I2C: IC_SS_SCL_HCNT Register

Offset: 0x14

Description

Standard Speed I2C Clock SCL High Count Register

Table 457.
IC_SS_SCL_HCNT
Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-

位	描述	类型	复位值
9	<p>STOP: 该位控制是否在数据字节发送或接收之后发出 STOP。</p> <p>- 1 - 无论 Tx FIFO 是否为空，此字节后均发出 STOP 信号。如果 Tx FIFO 非空，主设备将立即尝试通过发出 START 信号并竞争总线来启动新的传输。- 0 - 无论 Tx FIFO 是否为空，此字节后均不发出 STOP 信号。如果 Tx FIFO 非空，主设备根据 CMD 位的值继续通过发送/接收数据字节完成当前传输。如果 Tx FIFO 为空，主设备将保持 SCL 线为低电平，暂停总线，直到 Tx FIFO 中有新的命令。复位值：0x0</p>	SC	0x0
	枚举值：		
	0x0 → 禁用：此命令后不发出 STOP 信号		
	0x1 → 启用：此命令后发出 STOP 信号		
8	<p>CMD: 此位控制执行读操作或写操作。当 DW_apb_i2con 作为从设备时，此位不控制方向。仅在作为主设备时控制方向。</p> <p>当命令输入 TX FIFO 时，此位用于区分写命令和读命令。在从设备接收模式下，此位为“无关位”，因无需写入该寄存器。在从设备发送模式下，‘0’表示将传输 IC_DATA_CMD 中的数据。</p> <p>编程设置此位时，请注意：若在发送 General Call 命令后尝试执行读操作，除非已清除 IC_TAR 寄存器第 11 位（SPECIAL），否则会触发 TX_ABRT 中断（IC_RA_W_INTR_STAT 寄存器第 6 位）。若在接收到 RD_REQ 中断后向此位写入 ‘1’，则会引发 TX_ABRT 中断。</p> <p>复位值：0x0</p>	SC	0x0
	枚举值：		
	0x0 → WRITE：主设备写命令		
	0x1 → READ：主设备读命令		
7:0	<p>DAT: 该寄存器包含通过 I2C 总线传输或接收的数据。写入该寄存器并意图进行读操作时，DW_apb_i2c 会忽略位 7:0 (DAT)；读取该寄存器时，这些位返回 DW_apb_i2c 接口接收的数据值。</p> <p>复位值：0x0</p>	读写	0x00

I2C: IC_SS_SCL_HCNT 寄存器

偏移量: 0x14

描述

标准速度 I2C 时钟 SCL 高电平计数寄存器

表457。
IC_SS_SCL_HCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-

Bits	Description	Type	Reset
15:0	<p>IC_SS_SCL_HCNT: This register must be set before any I2C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock high-period count for standard speed. For more information, refer to 'IC_CLK Frequency Configuration'.</p> <p>This register can be written only when the I2C interface is disabled which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 6; hardware prevents values less than this being written, and if attempted results in 6 being set. For designs with APB_DATA_WIDTH = 8, the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed.</p> <p>NOTE: This register must not be programmed to a value higher than 65525, because DW_apb_i2c uses a 16-bit counter to flag an I2C bus idle condition when this counter reaches a value of IC_SS_SCL_HCNT + 10.</p>	RW	0x0028

I2C: IC_SS_SCL_LCNT Register

Offset: 0x18

Description

Standard Speed I2C Clock SCL Low Count Register

Table 458.
IC_SS_SCL_LCNT
Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	<p>IC_SS_SCL_LCNT: This register must be set before any I2C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock low period count for standard speed. For more information, refer to 'IC_CLK Frequency Configuration'.</p> <p>This register can be written only when the I2C interface is disabled which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 8; hardware prevents values less than this being written, and if attempted, results in 8 being set. For designs with APB_DATA_WIDTH = 8, the order of programming is important to ensure the correct operation of DW_apb_i2c. The lower byte must be programmed first, and then the upper byte is programmed.</p>	RW	0x002f

I2C: IC_FS_SCL_HCNT Register

Offset: 0x1c

Description

Fast Mode or Fast Mode Plus I2C Clock SCL High Count Register

Table 459.
IC_FS_SCL_HCNT
Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-

位	描述	类型	复位值
15:0	<p>IC_SS_SCL_HCNT: 必须在任何 I2C 总线事务开始之前设置此寄存器，以确保正确的输入/输出时序。该寄存器设置标准速率下 SCL 时钟的高电平周期计数。更多信息请参阅“IC_CLK 频率配置”。</p> <p>仅当 I2C 接口被禁用时（对应 IC_ENABLE[0] 寄存器设为 0），才允许写入此寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 6；硬件会禁止写入小于此值的数值，若尝试写入，将自动设置为 6。对于 APB_DATA_WIDTH = 8 的设计，编程顺序至关重要，以确保 DW_apb_i2c 的正确运行。必须先编程低字节。然后编程高字节。</p> <p>注意：该寄存器的设置值不得高于 65525，因为 DW_apb_i2c 使用 16 位计数器，当计数器达到 IC_SS_SCL_HCNT + 10 时，表示 I2C 总线处于空闲状态。</p>	读写	0x0028

I2C: IC_SS_SCL_LCNT 寄存器

偏移：0x18

说明

标准速度I2C时钟SCL低电平计数寄存器

表 458。
IC_SS_SCL_LCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	<p>IC_SS_SCL_LCNT: 此寄存器必须在任何 I2C 总线事务开始前设置，以确保正确的 I/O 时序。此寄存器设定标准速率下 SCL 时钟的低电平周期计数。详情请参阅“IC_CLK 频率配置”。</p> <p>仅当 I2C 接口被禁用时（对应 IC_ENABLE[0] 寄存器设为 0），才允许写入此寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 8；硬件禁止写入小于此值的数，若尝试写入，将自动设为 8。对于 APB_DATA_WIDTH = 8 的设计，编程顺序至关重要，以保证 DW_apb_i2c 的正确操作。必须先编程低字节，然后编程高字节。</p>	读写	0x002f

I2C: IC_FS_SCL_HCNT 寄存器

偏移：0x1c

说明

快速模式或增强快速模式I2C时钟SCL高电平计数寄存器

表 459。
IC_FS_SCL_HCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-

Bits	Description	Type	Reset
15:0	<p>IC_FS_SCL_HCNT: This register must be set before any I2C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock high-period count for fast mode or fast mode plus. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. For more information, refer to 'IC_CLK Frequency Configuration'.</p> <p>This register goes away and becomes read-only returning 0s if IC_MAX_SPEED_MODE = standard. This register can be written only when the I2C interface is disabled, which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 6; hardware prevents values less than this being written, and if attempted results in 6 being set. For designs with APB_DATA_WIDTH == 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed.</p>	RW	0x0006

I2C: IC_FS_SCL_LCNT Register

Offset: 0x20

Description

Fast Mode or Fast Mode Plus I2C Clock SCL Low Count Register

Table 460.
IC_FS_SCL_LCNT
Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	<p>IC_FS_SCL_LCNT: This register must be set before any I2C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock low period count for fast speed. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. For more information, refer to 'IC_CLK Frequency Configuration'.</p> <p>This register goes away and becomes read-only returning 0s if IC_MAX_SPEED_MODE = standard.</p> <p>This register can be written only when the I2C interface is disabled, which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 8; hardware prevents values less than this being written, and if attempted results in 8 being set. For designs with APB_DATA_WIDTH = 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed. If the value is less than 8 then the count value gets changed to 8.</p>	RW	0x000d

I2C: IC_INTR_STAT Register

Offset: 0x2c

Description

I2C Interrupt Status Register

Each bit in this register has a corresponding mask bit in the IC_INTR_MASK register. These bits are cleared by reading the matching interrupt clear register. The unmasked raw versions of these bits are available in the IC_RAW_INTR_STAT

位	描述	类型	复位值
15:0	<p>IC_FS_SCL_HCNT: 此寄存器必须在任何 I2C 总线事务开始前设置，以确保正确的 I/O 时序。此寄存器用于设置快速模式或快速模式增强的 SCL 时钟高电平周期计数。该寄存器应用于高速模式，用以发送主控代码及起始字节或通用呼叫。详情请参阅“IC_CLK 频率配置”。</p> <p>当 IC_MAX_SPEED_MODE 设为 standard 时，该寄存器将消失并变为只读，返回值恒为 0。仅当 I2C 接口被禁用（对应 IC_ENABLE[0] 寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 6；硬件禁止写入低于此值的数值，尝试写入时将自动设定为 6。对于 APB_DATA_WIDTH 为 8 的设计，编程顺序尤为关键，以确保 DW_apb_i2c 的正常运作，必须先写入低字节。然后编程高字节。</p>	读写	0x0006

I2C: IC_FS_SCL_LCNT 寄存器

偏移: 0x20

说明

快速模式或增强快速模式 I2C 时钟 SCL 低电平计数寄存器

表460。
IC_FS_SCL_LCNT
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	<p>IC_FS_SCL_LCNT: 在执行任何 I2C 总线传输之前，必须先设置此寄存器以保证正确的输入输出时序。此寄存器用于设置快速模式下 SCL 时钟低电平周期计数。该寄存器应用于高速模式，用以发送主控代码及起始字节或通用呼叫。详情请参阅“IC_CLK 频率配置”。</p> <p>当 IC_MAX_SPEED_MODE = standard 时，该寄存器将失效并变为只读，返回值为 0。</p> <p>仅当 I2C 接口被禁用（对应 IC_ENABLE[0] 寄存器置零）时，方可写入该寄存器。其他时间的写入操作无效。</p> <p>最小有效值为 8；硬件禁止写入小于此值的数值，若尝试写入，则会被设置为 8。对于 APB_DATA_WIDTH = 8 的设计，编程顺序极为重要，以确保 DW_apb_i2c 的正确操作。必须先编程低字节。然后编程高字节。若该值小于 8，则计数值会被更改为 8。</p>	读写	0x000d

I2C: IC_INTR_STAT 寄存器

偏移量: 0x2c

描述

I2C 中断状态寄存器

该寄存器的每一位都对应于 IC_INTR_MASK 寄存器中的掩码位。通过读取相应的中断清除寄存器，可清除这些位。这些位的未屏蔽原始版本可在 IC_RAW_INTR_STAT 寄存器中获取。

register.

*Table 461.
IC_INTR_STAT
Register*

Bits	Description	Type	Reset
31:13	Reserved.	-	-
12	R_RESTART_DET: See IC_RAW_INTR_STAT for a detailed description of R_RESTART_DET bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_RESTART_DET interrupt is inactive		
	0x1 → ACTIVE: R_RESTART_DET interrupt is active		
11	R_GEN_CALL: See IC_RAW_INTR_STAT for a detailed description of R_GEN_CALL bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_GEN_CALL interrupt is inactive		
	0x1 → ACTIVE: R_GEN_CALL interrupt is active		
10	R_START_DET: See IC_RAW_INTR_STAT for a detailed description of R_START_DET bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_START_DET interrupt is inactive		
	0x1 → ACTIVE: R_START_DET interrupt is active		
9	R_STOP_DET: See IC_RAW_INTR_STAT for a detailed description of R_STOP_DET bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_STOP_DET interrupt is inactive		
	0x1 → ACTIVE: R_STOP_DET interrupt is active		
8	R_ACTIVITY: See IC_RAW_INTR_STAT for a detailed description of R_ACTIVITY bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_ACTIVITY interrupt is inactive		
	0x1 → ACTIVE: R_ACTIVITY interrupt is active		
7	R_RX_DONE: See IC_RAW_INTR_STAT for a detailed description of R_RX_DONE bit. Reset value: 0x0	RO	0x0
	Enumerated values:		

寄存器的内容。

表 461。
IC_INTR_STAT
寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	R_RESTART_DET: 关于 R_RESTART_DET 位的详细描述, 请参见 IC_RAW_INT R_STAT 寄存器。 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_RESTART_DET 中断处于非激活状态		
	0x1 → 激活: R_RESTART_DET 中断处于激活状态		
11	R_GEN_CALL: 详见 IC_RAW_INTR_STAT 中 R_GEN_CALL 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_GEN_CALL 中断处于非激活状态		
	0x1 → 激活: R_GEN_CALL 中断处于激活状态		
10	R_START_DET: 详见 IC_RAW_INTR_STAT 中 R_START_DET 位的描述。 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_START_DET 中断处于非激活状态		
	0x1 → 激活: R_START_DET 中断处于激活状态		
9	R_STOP_DET: 详见 IC_RAW_INTR_STAT 中 R_STOP_DET 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_STOP_DET 中断处于非激活状态		
	0x1 → 激活: R_STOP_DET 中断处于激活状态		
8	R_ACTIVITY: 详见 IC_RAW_INTR_STAT 中 R_ACTIVITY 位的描述 ◦ 复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → 未激活: R_ACTIVITY 中断处于非激活状态		
	0x1 → 活动: R_ACTIVITY 中断处于活动状态		
7	R_RX_DONE: 详见 IC_RAW_INTR_STAT 中 R_RX_DONE 位的描述。 复位值: 0x0	只读	0x0
	枚举值:		

Bits	Description	Type	Reset
	0x0 → INACTIVE: R_RX_DONE interrupt is inactive		
	0x1 → ACTIVE: R_RX_DONE interrupt is active		
6	R_TX_ABRT : See IC_RAW_INTR_STAT for a detailed description of R_TX_ABRT bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_TX_ABRT interrupt is inactive		
	0x1 → ACTIVE: R_TX_ABRT interrupt is active		
5	R_RD_REQ : See IC_RAW_INTR_STAT for a detailed description of R_RD_REQ bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_RD_REQ interrupt is inactive		
	0x1 → ACTIVE: R_RD_REQ interrupt is active		
4	R_TX_EMPTY : See IC_RAW_INTR_STAT for a detailed description of R_TX_EMPTY bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_TX_EMPTY interrupt is inactive		
	0x1 → ACTIVE: R_TX_EMPTY interrupt is active		
3	R_TX_OVER : See IC_RAW_INTR_STAT for a detailed description of R_TX_OVER bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_TX_OVER interrupt is inactive		
	0x1 → ACTIVE: R_TX_OVER interrupt is active		
2	R_RX_FULL : See IC_RAW_INTR_STAT for a detailed description of R_RX_FULL bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: R_RX_FULL interrupt is inactive		
	0x1 → ACTIVE: R_RX_FULL interrupt is active		
1	R_RX_OVER : See IC_RAW_INTR_STAT for a detailed description of R_RX_OVER bit. Reset value: 0x0	RO	0x0
	Enumerated values:		

位	描述	类型	复位值
	0x0 → 非活动：R_RX_DONE中断处于非活动状态		
	0x1 → 活动：R_RX_DONE中断处于活动状态		
6	R_TX_ABRT : 详见IC_RAW_INTR_STAT中R_TX_ABRT位的描述。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_ABRT中断处于非活动状态		
	0x1 → 活动：R_TX_ABRT中断处于活动状态		
5	R_RD_REQ : 详见IC_RAW_INTR_STAT中R_RD_REQ位的描述。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_RD_REQ中断处于非活动状态		
	0x1 → 活动：R_RD_REQ中断处于活动状态		
4	R_TX_EMPTY : 详见IC_RAW_INTR_STAT中R_TX_EMPTY位的描述。 。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_EMPTY中断处于非活动状态		
	0x1 → 活动：R_TX_EMPTY中断处于活动状态		
3	R_TX_OVER : 有关R_TX_OVER位的详细说明，请参见IC_RAW_INTR_STAT。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_TX_OVER中断处于非活动状态		
	0x1 → 活动：R_TX_OVER中断处于活动状态		
2	R_RX_FULL : 有关R_RX_FULL位的详细说明，请参见IC_RAW_INTR_STAT。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：R_RX_FULL中断处于非活动状态		
	0x1 → 活动：R_RX_FULL中断处于活动状态		
1	R_RX_OVER : 有关R_RX_OVER位的详细说明，请参见IC_RAW_INT_R_STAT。 复位值：0x0	只读	0x0
	枚举值：		

Bits	Description	Type	Reset
	0x0 → INACTIVE: R_RX_OVER interrupt is inactive		
	0x1 → ACTIVE: R_RX_OVER interrupt is active		
0	R_RX_UNDER : See IC_RAW_INTR_STAT for a detailed description of R_RX_UNDER bit. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RX_UNDER interrupt is inactive		
	0x1 → ACTIVE: RX_UNDER interrupt is active		

I2C: IC_INTR_MASK Register

Offset: 0x30

Description

I2C Interrupt Mask Register.

These bits mask their corresponding interrupt status bits. This register is active low; a value of 0 masks the interrupt, whereas a value of 1 unmasks the interrupt.

Table 462.
IC_INTR_MASK
Register

Bits	Description	Type	Reset
31:13	Reserved.	-	-
12	M_RESTART_DET : This bit masks the R_RESTART_DET interrupt in IC_INTR_STAT register. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → ENABLED: RESTART_DET interrupt is masked		
	0x1 → DISABLED: RESTART_DET interrupt is unmasked		
11	M_GEN_CALL : This bit masks the R_GEN_CALL interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: GEN_CALL interrupt is masked		
	0x1 → DISABLED: GEN_CALL interrupt is unmasked		
10	M_START_DET : This bit masks the R_START_DET interrupt in IC_INTR_STAT register. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → ENABLED: START_DET interrupt is masked		
	0x1 → DISABLED: START_DET interrupt is unmasked		

位	描述	类型	复位值
	0x0 → 非活动：R_RX_OVER中断处于非活动状态		
	0x1 → 活动：R_RX_OVER中断处于活动状态		
0	R_RX_UNDER : 有关R_RX_UNDER位的详细说明，请参见IC_RAW_INTR_STAT。 复位值：0x0	只读	0x0
	枚举值：		
	0x0 → 非活动：RX_UNDER中断处于非活动状态		
	0x1 → 活动：RX_UNDER中断处于活动状态		

I2C: IC_INTR_MASK寄存器

偏移：0x30

描述

I2C 中断屏蔽寄存器。

这些位用于屏蔽其对应的中断状态位。该寄存器为低电平有效；值为0时屏蔽中断，值为1时取消屏蔽中断。

表462。
IC_INTR_MASK
寄存器

位	描述	类型	复位值
31:13	保留。	-	-
12	M_RESTART_DET : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_RESTART_DET 中断。 IC_INTR_STAT 寄存器。 复位值：0x0	读写	0x0
	枚举值：		
	0x0 → 启用：RESTART_DET 中断被屏蔽		
	0x1 → 禁用：RESTART_DET 中断未被屏蔽		
11	M_GEN_CALL : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_GEN_CALL 中断。 复位值：0x1	读写	0x1
	枚举值：		
	0x0 → 启用：GEN_CALL 中断被屏蔽		
	0x1 → 禁用：GEN_CALL 中断未被屏蔽		
10	M_START_DET : 该位屏蔽 IC_INTR_STAT 寄存器中的 R_START_DET 中断。 复位值：0x0	读写	0x0
	枚举值：		
	0x0 → 启用：START_DET 中断被屏蔽		
	0x1 → 禁用：START_DET 中断未被屏蔽		

Bits	Description	Type	Reset
9	M_STOP_DET: This bit masks the R_STOP_DET interrupt in IC_INTR_STAT register. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → ENABLED: STOP_DET interrupt is masked		
	0x1 → DISABLED: STOP_DET interrupt is unmasked		
8	M_ACTIVITY: This bit masks the R_ACTIVITY interrupt in IC_INTR_STAT register. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → ENABLED: ACTIVITY interrupt is masked		
	0x1 → DISABLED: ACTIVITY interrupt is unmasked		
7	M_RX_DONE: This bit masks the R_RX_DONE interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: RX_DONE interrupt is masked		
	0x1 → DISABLED: RX_DONE interrupt is unmasked		
6	M_TX_ABRT: This bit masks the R_TX_ABRT interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: TX_ABORT interrupt is masked		
	0x1 → DISABLED: TX_ABORT interrupt is unmasked		
5	M_RD_REQ: This bit masks the R_RD_REQ interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: RD_REQ interrupt is masked		
	0x1 → DISABLED: RD_REQ interrupt is unmasked		
4	M_TX_EMPTY: This bit masks the R_TX_EMPTY interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: TX_EMPTY interrupt is masked		
	0x1 → DISABLED: TX_EMPTY interrupt is unmasked		

位	描述	类型	复位值
9	M_STOP_DET: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_STOP_DET 中断。 复位值: 0x0	读写	0x0
	枚举值:		
	0x0 → 已启用: STOP_DET 中断被屏蔽		
	0x1 → 已禁用: STOP_DET 中断未被屏蔽		
8	M_ACTIVITY: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_ACTIVITY 中断。 复位值: 0x0	读写	0x0
	枚举值:		
	0x0 → 已启用: ACTIVITY 中断被屏蔽		
	0x1 → 已禁用: ACTIVITY 中断未被屏蔽		
7	M_RX_DONE: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_DONE 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RX_DONE 中断被屏蔽		
	0x1 → 已禁用: RX_DONE 中断未被屏蔽		
6	M_TX_ABRT: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_ABRT 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_ABORT 中断被屏蔽		
	0x1 → 已禁用: TX_ABORT 中断未被屏蔽		
5	M_RD_REQ: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RD_REQ 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RD_REQ 中断被屏蔽		
	0x1 → 已禁用: RD_REQ 中断未被屏蔽		
4	M_TX_EMPTY: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_EMPTY 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_EMPTY 中断被屏蔽		
	0x1 → 已禁用: TX_EMPTY 中断未被屏蔽		

Bits	Description	Type	Reset
3	M_TX_OVER: This bit masks the R_TX_OVER interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: TX_OVER interrupt is masked		
	0x1 → DISABLED: TX_OVER interrupt is unmasked		
2	M_RX_FULL: This bit masks the R_RX_FULL interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: RX_FULL interrupt is masked		
	0x1 → DISABLED: RX_FULL interrupt is unmasked		
1	M_RX_OVER: This bit masks the R_RX_OVER interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: RX_OVER interrupt is masked		
	0x1 → DISABLED: RX_OVER interrupt is unmasked		
0	M_RX_UNDER: This bit masks the R_RX_UNDER interrupt in IC_INTR_STAT register. Reset value: 0x1	RW	0x1
	Enumerated values:		
	0x0 → ENABLED: RX_UNDER interrupt is masked		
	0x1 → DISABLED: RX_UNDER interrupt is unmasked		

I2C: IC_RAW_INTR_STAT Register

Offset: 0x34

Description

I2C Raw Interrupt Status Register

Unlike the IC_INTR_STAT register, these bits are not masked so they always show the true status of the DW_apb_i2c.

Table 463.
IC_RAW_INTR_STAT
Register

Bits	Description	Type	Reset
31:13	Reserved.	-	-

位	描述	类型	复位值
3	M_TX_OVER: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_TX_OVER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: TX_OVER 中断被屏蔽		
	0x1 → 已禁用: TX_OVER 中断未被屏蔽		
2	M_RX_FULL: 此位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_FULL 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 已启用: RX_FULL 中断被屏蔽		
	0x1 → 已禁用: RX_FULL 中断未被屏蔽		
1	M_RX_OVER: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_OVER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 启用: RX_OVER 中断被屏蔽		
	0x1 → 禁用: RX_OVER 中断未被屏蔽		
0	M_RX_UNDER: 该位用于屏蔽 IC_INTR_STAT 寄存器中的 R_RX_UNDER 中断。 复位值: 0x1	读写	0x1
	枚举值:		
	0x0 → 启用: RX_UNDER 中断被屏蔽		
	0x1 → 禁用: RX_UNDER 中断未被屏蔽		

I2C: IC_RAW_INTR_STAT 寄存器

偏移: 0x34

描述

I2C原始中断状态寄存器

不同于 IC_INTR_STAT 寄存器，此处的位未被屏蔽，因此始终显示 DW_apb_i2c 的真实状态。

表 463。
IC_RAW_INTR_STAT
寄存器

位	描述	类型	复位值
31:13	保留。	-	-

Bits	Description	Type	Reset
12	<p>RESTART_DET: Indicates whether a RESTART condition has occurred on the I2C interface when DW_apb_i2c is operating in Slave mode and the slave is being addressed. Enabled only when IC_SLV_RESTART_DET_EN=1.</p> <p>Note: However, in high-speed mode or during a START BYTE transfer, the RESTART comes before the address field as per the I2C protocol. In this case, the slave is not the addressed slave when the RESTART is issued, therefore DW_apb_i2c does not generate the RESTART_DET interrupt.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RESTART_DET interrupt is inactive		
	0x1 → ACTIVE: RESTART_DET interrupt is active		
11	<p>GEN_CALL: Set only when a General Call address is received and it is acknowledged. It stays set until it is cleared either by disabling DW_apb_i2c or when the CPU reads bit 0 of the IC_CLR_GEN_CALL register. DW_apb_i2c stores the received data in the Rx buffer.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: GEN_CALL interrupt is inactive		
	0x1 → ACTIVE: GEN_CALL interrupt is active		
10	<p>START_DET: Indicates whether a START or RESTART condition has occurred on the I2C interface regardless of whether DW_apb_i2c is operating in slave or master mode.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: START_DET interrupt is inactive		
	0x1 → ACTIVE: START_DET interrupt is active		
9	<p>STOP_DET: Indicates whether a STOP condition has occurred on the I2C interface regardless of whether DW_apb_i2c is operating in slave or master mode.</p> <p>In Slave Mode: - If IC_CON[7]=1'b1 (STOP_DET_IFADDRESSED), the STOP_DET interrupt will be issued only if slave is addressed. Note: During a general call address, this slave does not issue a STOP_DET interrupt if STOP_DET_IF_ADDRESSED=1'b1, even if the slave responds to the general call address by generating ACK. The STOP_DET interrupt is generated only when the transmitted address matches the slave address (SAR). - If IC_CON[7]=1'b0 (STOP_DET_IFADDRESSED), the STOP_DET interrupt is issued irrespective of whether it is being addressed. In Master Mode: - If IC_CON[10]=1'b1 (STOP_DET_IF_MASTER_ACTIVE), the STOP_DET interrupt will be issued only if Master is active. - If IC_CON[10]=1'b0 (STOP_DET_IFADDRESSED), the STOP_DET interrupt will be issued irrespective of whether master is active or not. Reset value: 0x0</p>	RO	0x0
	Enumerated values:		

位	描述	类型	复位值
12	<p>RESTART_DET: 指示在 DW_apb_i2c 作为从机模式且从机被寻址时, I2C 接口是否发生了 RESTART 条件。仅当 IC_SLV_RESTART_DET_EN=1 时启用。</p> <p>注意: 但根据I2C协议, 在高速模式或传输START BYTE期间, RESTART 应位于地址字段之前。在此情况下, 当发出RESTART时, 被寻址的从机并非当前从机, 因此DW_apb_i2c不会产生RESTART_DET中断。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: RESTART_DET 中断处于非活动状态		
	0x1 → 活动: RESTART_DET 中断处于活动状态		
11	<p>GEN_CALL: 仅在接收到通用呼叫地址并确认时设置。该标志保持设置, 直到通过禁用DW_apb_i2c或CPU读取IC_CLR_GEN_CALL寄存器的第0位将其清除。DW_apb_i2c将接收的数据存储于接收缓冲区 (Rx buffer) 中。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: GEN_CALL 中断处于非活动状态		
	0x1 → 活动: GEN_CALL 中断处于活动状态		
10	<p>START_DET: 指示无论DW_apb_i2c处于从机还是主机模式, I2C接口上是否出现START或RESTART条件。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: START_DET 中断处于非活动状态		
	0x1 → 活动: START_DET 中断处于活动状态		
9	<p>STOP_DET: 指示无论 DW_apb_i2c 处于从模式还是主模式, I2C 接口上是否发生了停止条件。</p> <p>在从模式下: - 当 IC_CON[7]=1'b1 (STOP_DET_IFADDRESSED) 时, 仅在从设备被寻址时触发 STOP_DET 中断。注意: 在通用调用地址期间, 若 STOP_DET_IF_ADDRESSED=1'b1, 即使从设备通过响应通用调用地址并产生 ACK, 从设备也不会触发 STOP_DET 中断。仅当传输地址与从设备地址 (SAR) 匹配时, 才会触发 STOP_DET 中断。- 当 IC_CON[7]=1'b0 (STOP_DET_IFAD DRESSED) 时, 无论是否被寻址, 均会触发 STOP_DET 中断。在主模式下: - 当 IC_CON[10]=1'b1 (STOP_DET_IF_MASTER_ACTIVE) 时, 仅当主设备处于活动状态时才触发 STOP_DET 中断。- 若 IC_CON[10]=1'b0 (STOP_ DET_IFADDRESSED) , 则无论主控是否处于活动状态, 均会触发 ST OP_DET 中断。复位值: 0x0</p>	只读	0x0
	枚举值:		

Bits	Description	Type	Reset
	0x0 → INACTIVE: STOP_DET interrupt is inactive		
	0x1 → ACTIVE: STOP_DET interrupt is active		
8	ACTIVITY: This bit captures DW_apb_i2c activity and stays set until it is cleared. There are four ways to clear it: - Disabling the DW_apb_i2c - Reading the IC_CLR_ACTIVITY register - Reading the IC_CLR_INTR register - System reset Once this bit is set, it stays set unless one of the four methods is used to clear it. Even if the DW_apb_i2c module is idle, this bit remains set until cleared, indicating that there was activity on the bus. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RAW_INTR_ACTIVITY interrupt is inactive		
	0x1 → ACTIVE: RAW_INTR_ACTIVITY interrupt is active		
7	RX_DONE: When the DW_apb_i2c is acting as a slave-transmitter, this bit is set to 1 if the master does not acknowledge a transmitted byte. This occurs on the last byte of the transmission, indicating that the transmission is done. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RX_DONE interrupt is inactive		
	0x1 → ACTIVE: RX_DONE interrupt is active		
6	TX_ABRT: This bit indicates if DW_apb_i2c, as an I2C transmitter, is unable to complete the intended actions on the contents of the transmit FIFO. This situation can occur both as an I2C master or an I2C slave, and is referred to as a 'transmit abort'. When this bit is set to 1, the IC_TX_ABRT_SOURCE register indicates the reason why the transmit abort takes places. Note: The DW_apb_i2c flushes/resets/empties the TX_FIFO and RX_FIFO whenever there is a transmit abort caused by any of the events tracked by the IC_TX_ABRT_SOURCE register. The FIFOs remains in this flushed state until the register IC_CLR_TX_ABRT is read. Once this read is performed, the Tx FIFO is then ready to accept more data bytes from the APB interface. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: TX_ABRT interrupt is inactive		
	0x1 → ACTIVE: TX_ABRT interrupt is active		
5	RD_REQ: This bit is set to 1 when DW_apb_i2c is acting as a slave and another I2C master is attempting to read data from DW_apb_i2c. The DW_apb_i2c holds the I2C bus in a wait state (SCL=0) until this interrupt is serviced, which means that the slave has been addressed by a remote master that is asking for data to be transferred. The processor must respond to this interrupt and then write the requested data to the IC_DATA_CMD register. This bit is set to 0 just after the processor reads the IC_CLR_RD_REQ register. Reset value: 0x0	RO	0x0

位	描述	类型	复位值
	0x0 → 非活动：STOP_DET 中断处于非活动状态		
	0x1 → 活动：STOP_DET 中断处于活动状态		
8	<p>ACTIVITY：该位记录 DW_apb_i2c 活动信号，并保持置位直至被清除。清除该位的方法有四种：</p> <ul style="list-style-type: none"> - 禁用 DW_apb_i2c - 读取 IC_CLR_ACTIVITY 寄存器 - 读取 IC_CLR_INTR 寄存器 - 系统复位 - 一旦该位被置位，除非通过上述任一方法进行清除，否则将持续保持置位。即便 DW_apb_i2c 模块空闲，该位亦表明总线曾有活动且保持置位状态。 <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：RAW_INTR_ACTIVITY 中断处于非活动状态		
	0x1 → 活动：RAW_INTR_ACTIVITY 中断处于活动状态		
7	<p>RX_DONE：当 DW_apb_i2c 作为从设备发送方时，如果主设备未确认所发送的字节，该位将被置为1。此状态发生在传输的最后一个字节，表示传输已完成。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：RX_DONE 中断处于非活动状态。		
	0x1 → 活动：RX_DONE 中断处于活动状态。		
6	<p>TX_ABRT：该位指示 DW_apb_i2c 作为 I2C 发送方时，无法完成对发送 FIFO 内容的预期操作。此情况可能发生在 I2C 主设备或从设备，称为“发送中止”。当该位被置为1时，IC_TX_ABRT_SOURCE 寄存器会指示发送中止的具体原因。</p> <p>注意：无论导致发送中止的事件属于 IC_TX_ABRT_SOURCE 寄存器所跟踪的哪种情况，DW_apb_i2c 都会清空、重置或清除 TX_FIFO 和 RX_FIFO。FIFO 将保持在此清空状态，直到读取寄存器 IC_CLR_TX_ABRT。一旦执行该读取操作，Tx FIFO 即准备好接受来自 APB 接口的更多数据字节。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非激活：TX_ABRT 中断处于非激活状态		
	0x1 → 激活：TX_ABRT 中断处于激活状态		
5	<p>RD_REQ：当 DW_apb_i2c 作为从设备且另一 I2C 主设备试图从 DW_apb_i2c 读取数据时，该位被置为1。DW_apb_i2c 会将 I2C 总线保持在等待状态 (SC L=0)，直到该中断被处理，这意味着从设备已被远程主设备寻址，且主设备请求进行数据传输。处理器必须响应该中断，然后将请求的数据写入 IC_DARTA_CMD 寄存器。该位在处理器读取 IC_CLR_RD_REQ 寄存器后立即被清零。</p> <p>复位值：0x0</p>	只读	0x0

Bits	Description	Type	Reset
	Enumerated values:		
	0x0 → INACTIVE: RD_REQ interrupt is inactive		
	0x1 → ACTIVE: RD_REQ interrupt is active		
4	<p>TX_EMPTY: The behavior of the TX_EMPTY interrupt status differs based on the TX_EMPTY_CTRL selection in the IC_CON register.</p> <ul style="list-style-type: none"> - When TX_EMPTY_CTRL = 0: This bit is set to 1 when the transmit buffer is at or below the threshold value set in the IC_TX_TL register. - When TX_EMPTY_CTRL = 1: This bit is set to 1 when the transmit buffer is at or below the threshold value set in the IC_TX_TL register and the transmission of the address/data from the internal shift register for the most recently popped command is completed. It is automatically cleared by hardware when the buffer level goes above the threshold. When IC_ENABLE[0] is set to 0, the TX FIFO is flushed and held in reset. There the TX FIFO looks like it has no data within it, so this bit is set to 1, provided there is activity in the master or slave state machines. When there is no longer any activity, then with ic_en=0, this bit is set to 0. <p>Reset value: 0x0.</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: TX_EMPTY interrupt is inactive		
	0x1 → ACTIVE: TX_EMPTY interrupt is active		
3	<p>TX_OVER: Set during transmit if the transmit buffer is filled to IC_TX_BUFFER_DEPTH and the processor attempts to issue another I2C command by writing to the IC_DATA_CMD register. When the module is disabled, this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: TX_OVER interrupt is inactive		
	0x1 → ACTIVE: TX_OVER interrupt is active		
2	<p>RX_FULL: Set when the receive buffer reaches or goes above the RX_TL threshold in the IC_RX_TL register. It is automatically cleared by hardware when buffer level goes below the threshold. If the module is disabled (IC_ENABLE[0]=0), the RX FIFO is flushed and held in reset; therefore the RX FIFO is not full. So this bit is cleared once the IC_ENABLE bit 0 is programmed with a 0, regardless of the activity that continues.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RX_FULL interrupt is inactive		
	0x1 → ACTIVE: RX_FULL interrupt is active		

位	描述	类型	复位值
	枚举值：		
	0x0 → 非活动： RD_REQ 中断处于非活动状态		
	0x1 → 活动： RD_REQ 中断处于活动状态		
4	<p>TX_EMPTY: TX_EMPTY 中断状态的行为因 IC_CON 寄存器中 TX_EMPTY_C_TRL 的设置而异。 - 当 TX_EMPTY_CTRL = 0 时：当发射缓冲区中数据量小于或等于 IC_TX_TL 寄存器中设置的阈值时，该位被置为 1 。 - 当 TX_EMPTY_CTRL = 1 时：当发射缓冲区中数据量小于或等于 IC_TX_TL 寄存器中设置的阈值，且最近出栈命令的内部移位寄存器中的地址/数据传输完成时，该位被置为 1。当缓冲区数据量高于阈值时，硬件会自动清除此位。当 IC_ENABLE[0] 设置为 0 时，TX FIFO 会被清空并保持复位状态。此时，TX FIFO 似乎无数据，在主机或从机状态机存在活动时，该位被置为 1。当不再有任何活动且 ic_en=0 时，此位被置为 0。</p> <p>复位值：0x0。</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： TX_EMPTY 中断处于非活动状态		
	0x1 → 活动： TX_EMPTY 中断处于活动状态		
3	<p>TX_OVER: 当传输缓冲区填满至 IC_TX_BUFFER_DEPTH 且处理器尝试通过写入 IC_DATA_CMD 寄存器发出另一个 I2C 命令时，在传输过程中设置该位。当模块被禁用时，此位保持其状态，直到主机或从机状态机进入空闲状态；当 ic_en 变为 0 时，此中断被清除。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： TX_OVER 中断处于非活动状态		
	0x1 → 活动： TX_OVER 中断处于活动状态		
2	<p>RX_FULL: 当接收缓冲区达到或超过 IC_RX_TL 寄存器中 RX_TL 阈值时设置。当缓冲区电平降至阈值以下时，硬件会自动清除此位。如果模块被禁用 (IC_ENABLE[0]=0) ，则 RX FIFO 将被清空并保持复位状态；因此 RX FIFO 未满。因此，一旦 IC_ENABLE 位 0 被设置为 0，无论后续活动如何，该位即被清除。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动： RX_FULL 中断处于非激活状态		
	0x1 → 激活： RX_FULL 中断处于激活状态		

Bits	Description	Type	Reset
1	<p>RX_OVER: Set if the receive buffer is completely filled to IC_RX_BUFFER_DEPTH and an additional byte is received from an external I2C device. The DW_apb_i2c acknowledges this, but any data bytes received after the FIFO is full are lost. If the module is disabled (IC_ENABLE[0]=0), this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>Note: If bit 9 of the IC_CON register (RX_FIFO_FULL_HLD_CTRL) is programmed to HIGH, then the RX_OVER interrupt never occurs, because the Rx FIFO never overflows.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RX_OVER interrupt is inactive		
	0x1 → ACTIVE: RX_OVER interrupt is active		
0	<p>RX_UNDER: Set if the processor attempts to read the receive buffer when it is empty by reading from the IC_DATA_CMD register. If the module is disabled (IC_ENABLE[0]=0), this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: RX_UNDER interrupt is inactive		
	0x1 → ACTIVE: RX_UNDER interrupt is active		

I2C: IC_RX_TL Register

Offset: 0x38

Description

I2C Receive FIFO Threshold Register

Table 464. IC_RX_TL Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	<p>RX_TL: Receive FIFO Threshold Level.</p> <p>Controls the level of entries (or above) that triggers the RX_FULL interrupt (bit 2 in IC_RAW_INTR_STAT register). The valid range is 0-255, with the additional restriction that hardware does not allow this value to be set to a value larger than the depth of the buffer. If an attempt is made to do that, the actual value set will be the maximum depth of the buffer. A value of 0 sets the threshold for 1 entry, and a value of 255 sets the threshold for 256 entries.</p>	RW	0x00

I2C: IC_TX_TL Register

Offset: 0x3c

Description

I2C Transmit FIFO Threshold Register

位	描述	类型	复位值
1	<p>RX_OVER: 当接收缓冲区完全填满, 且达到 IC_RX_BUFFER_DEPTH 后, 再从外部 I2C 设备接收额外字节时设置。DW_apb_i2c 会对此进行应答, 但 FIFO 满后接收的任何数据字节将丢失。如果模块被禁用 (IC_ENABLE[0]=0), 该位保持其状态, 直至主机或从机状态机进入空闲状态, 并且当 ic_en 变为 0 时, 该中断被清除。</p> <p>注意: 若 IC_CON 寄存器的第 9 位 (RX_FIFO_FULL_HLD_CTRL) 设为高电平, 则 RX_OVER 中断不会发生, 因为 Rx FIFO 永远不会溢出。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非激活: RX_OVER 中断处于非激活状态		
	0x1 → 激活: RX_OVER 中断处于激活状态		
0	<p>RX_UNDER: 当处理器尝试通过读取 IC_DATA_CMD 寄存器读取空的接收缓冲区时设置该标志。如果模块被禁用 (IC_ENABLE[0]=0), 该位保持其状态, 直至主机或从机状态机进入空闲状态, 并且当 ic_en 变为 0 时, 该中断被清除。</p> <p>复位值: 0x0</p>	只读	0x0
	枚举值:		
	0x0 → 非活动: RX_UNDER 中断处于非活动状态		
	0x1 → 活动: RX_UNDER 中断处于活动状态		

I2C: IC_RX_TL 寄存器

偏移: 0x38

描述

I2C接收FIFO阈值寄存器

表 464. IC_RX_TL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	<p>RX_TL: 接收 FIFO 阈值等级。</p> <p>控制触发 RX_FULL 中断 (IC_RAW_INTR_STAT 寄存器第 2 位) 所需的条目数 (及以上)。有效取值范围为 0-255, 且硬件不允许将该值设置为超过缓冲区深度的数值。若尝试设置超过缓冲区深度的值, 实际生效值将为缓冲区的最大深度。取值为 0 时, 阈值为 1 个条目; 取值为 255 时, 阈值为 256 个条目。</p>	读写	0x00

I2C: IC_TX_TL 寄存器

偏移: 0x3c

描述

I2C发送FIFO阈值寄存器

Table 465. IC_TX_TL Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	TX_TL : Transmit FIFO Threshold Level. Controls the level of entries (or below) that trigger the TX_EMPTY interrupt (bit 4 in IC_RAW_INTR_STAT register). The valid range is 0-255, with the additional restriction that it may not be set to value larger than the depth of the buffer. If an attempt is made to do that, the actual value set will be the maximum depth of the buffer. A value of 0 sets the threshold for 0 entries, and a value of 255 sets the threshold for 255 entries.	RW	0x00

I2C: IC_CLR_INTR Register

Offset: 0x40

Description

Clear Combined and Individual Interrupt Register

Table 466. IC_CLR_INTR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_INTR : Read this register to clear the combined interrupt, all individual interrupts, and the IC_TX_ABRT_SOURCE register. This bit does not clear hardware clearable interrupts but software clearable interrupts. Refer to Bit 9 of the IC_TX_ABRT_SOURCE register for an exception to clearing IC_TX_ABRT_SOURCE. Reset value: 0x0	RO	0x0

I2C: IC_CLR_RX_UNDER Register

Offset: 0x44

Description

Clear RX_UNDER Interrupt Register

Table 467. IC_CLR_RX_UNDER Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_RX_UNDER : Read this register to clear the RX_UNDER interrupt (bit 0) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_RX_OVER Register

Offset: 0x48

Description

Clear RX_OVER Interrupt Register

Table 468. IC_CLR_RX_OVER Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-

表465. IC_TX_TL
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TX_TL : 发送FIFO阈值级别。 控制触发TX_EMPTY中断 (IC_RAW_INTR_STAT寄存器第4位) 的条目数阈值 (等于或低于该阈值)。有效范围为0至255，且不可设置为超过缓冲区深度的值。若尝试设置超过缓冲区深度的值，实际生效值将为缓冲区的最大深度。设置为0表示阈值为0条目，设置为255表示阈值为255条目。	读写	0x00

I2C: IC_CLR_INTR寄存器

偏移: 0x40

描述

清除组合中断及单个中断寄存器

表466.
IC_CLR_INTR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_INTR : 读取此寄存器以清除组合中断、所有单个中断，以及IC_TX_ABRT_SOURCE寄存器。此位不会清除硬件可清除的中断，仅清除软件可清除的中断。清除IC_TX_ABRT_SOURCE的例外情况详见IC_TX_ABRT_SOURCE寄存器的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_UNDER寄存器

偏移: 0x44

描述

清除 RX_UNDER 中断寄存器

表467.
IC_CLR_RX_UNDER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RX_UNDER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_UNDER 中断 (第0位)。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_OVER寄存器

偏移: 0x48

描述

清除 RX_OVER 中断寄存器

表468.
IC_CLR_RX_OVER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-

Bits	Description	Type	Reset
0	CLR_RX_OVER: Read this register to clear the RX_OVER interrupt (bit 1) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_TX_OVER Register

Offset: 0x4c

Description

Clear TX_OVER Interrupt Register

Table 469.
IC_CLR_TX_OVER
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_TX_OVER: Read this register to clear the TX_OVER interrupt (bit 3) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_RD_REQ Register

Offset: 0x50

Description

Clear RD_REQ Interrupt Register

Table 470.
IC_CLR_RD_REQ
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_RD_REQ: Read this register to clear the RD_REQ interrupt (bit 5) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_TX_ABRT Register

Offset: 0x54

Description

Clear TX_ABRT Interrupt Register

位	描述	类型	复位值
0	CLR_RX_OVER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_OVER 中断（第1位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_TX_OVER 寄存器

偏移量: 0x4c

说明

清除 TX_OVER 中断寄存器

表469。
IC_CLR_TX_OVER
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_TX_OVER : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 TX_OVER 中断（第3位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RD_REQ 寄存器

偏移: 0x50

说明

清除 RD_REQ 中断寄存器

表470。
IC_CLR_RD_REQ
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RD_REQ : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RD_REQ 中断（第5位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_TX_ABRT 寄存器

偏移: 0x54

说明

清除 TX_ABRT 中断寄存器

Table 471.
IC_CLR_TX_ABRT
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_TX_ABRT : Read this register to clear the TX_ABRT interrupt (bit 6) of the IC_RAW_INTR_STAT register, and the IC_TX_ABRT_SOURCE register. This also releases the TX FIFO from the flushed/reset state, allowing more writes to the TX FIFO. Refer to Bit 9 of the IC_TX_ABRT_SOURCE register for an exception to clearing IC_TX_ABRT_SOURCE. Reset value: 0x0	RO	0x0

I2C: IC_CLR_RX_DONE Register

Offset: 0x58

Description

Clear RX_DONE Interrupt Register

Table 472.
IC_CLR_RX_DONE
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_RX_DONE : Read this register to clear the RX_DONE interrupt (bit 7) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_ACTIVITY Register

Offset: 0x5c

Description

Clear ACTIVITY Interrupt Register

Table 473.
IC_CLR_ACTIVITY
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_ACTIVITY : Reading this register clears the ACTIVITY interrupt if the I2C is not active anymore. If the I2C module is still active on the bus, the ACTIVITY interrupt bit continues to be set. It is automatically cleared by hardware if the module is disabled and if there is no further activity on the bus. The value read from this register to get status of the ACTIVITY interrupt (bit 8) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_STOP_DET Register

Offset: 0x60

Description

Clear STOP_DET Interrupt Register

Table 474.
IC_CLR_STOP_DET
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-

表471。
IC_CLR_TX_ABRT
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_TX_ABRT : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 TX_ABRT 中断（第6位）及 IC_TX_ABRT_SOURCE 寄存器的状态。这也会解除 TX FIFO 的刷新/复位状态，允许向 TX FIFO 进行更多写入。有关不清除 IC_TX_ABRT_SOURCE 的例外情况，请参阅 IC_TX_ABRT_SOURCE 寄存器的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_RX_DONE 寄存器

偏移: 0x58

说明

清除 RX_DONE 中断寄存器

表472。
IC_CLR_RX_DONE
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RX_DONE : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 RX_DONE 中断（第7位）。 复位值: 0x0	只读	0x0

I2C: IC_CLR_ACTIVITY 寄存器

偏移: 0x5c

描述

清除 ACTIVITY 中断寄存器

表473。
IC_CLR_ACTIVITY
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_ACTIVITY : 读取此寄存器以清除当 I2C 总线不再活动时的 ACTIVITY 中断。若 I2C 模块仍在总线上处于活动状态，ACTIVITY 中断位将继续置位。该中断位在模块被禁用且总线上无进一步活动时，会由硬件自动清除。读取此寄存器可获取 IC_RAW_INTR_STAT 寄存器中 ACTIVITY 中断（第8位）的状态。 复位值: 0x0	只读	0x0

I2C: IC_CLR_STOP_DET 寄存器

偏移: 0x60

描述

清除 STOP_DET 中断寄存器

表474。
IC_CLR_STOP_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-

Bits	Description	Type	Reset
0	CLR_STOP_DET : Read this register to clear the STOP_DET interrupt (bit 9) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_START_DET Register

Offset: 0x64

Description

Clear START_DET Interrupt Register

Table 475.
IC_CLR_START_DET
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_START_DET : Read this register to clear the START_DET interrupt (bit 10) of the IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_CLR_GEN_CALL Register

Offset: 0x68

Description

Clear GEN_CALL Interrupt Register

Table 476.
IC_CLR_GEN_CALL
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_GEN_CALL : Read this register to clear the GEN_CALL interrupt (bit 11) of IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_ENABLE Register

Offset: 0x6c

Description

I2C Enable Register

Table 477. IC_ENABLE
Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	TX_CMD_BLOCK : In Master mode: - 1'b1: Blocks the transmission of data on I2C bus even if Tx FIFO has data to transmit. - 1'b0: The transmission of data starts on I2C bus automatically, as soon as the first data is available in the Tx FIFO. Note: To block the execution of Master commands, set the TX_CMD_BLOCK bit only when Tx FIFO is empty (IC_STATUS[2]==1) and Master is in Idle state (IC_STATUS[5] == 0). Any further commands put in the Tx FIFO are not executed until TX_CMD_BLOCK bit is unset. Reset value: IC_TX_CMD_BLOCK_DEFAULT	RW	0x0
	Enumerated values:		
	0x0 → NOT_BLOCKED: Tx Command execution not blocked		

位	描述	类型	复位值
0	CLR_STOP_DET : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 STOP_DET 中断的第9位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_START_DET 寄存器

偏移: 0x64

描述

清除 START_DET 中断寄存器

表475。
IC_CLR_START_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_START_DET : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 START_DET 中断的第10位。 复位值: 0x0	只读	0x0

I2C: IC_CLR_GEN_CALL 寄存器

偏移: 0x68

描述

清除 GEN_CALL 中断寄存器

表476。
IC_CLR_GEN_CALL
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_GEN_CALL : 读取此寄存器以清除 IC_RAW_INTR_STAT 寄存器中 GEN_CALL 中断的第11位。 复位值: 0x0	只读	0x0

I2C: IC_ENABLE 寄存器

偏移: 0x6c

描述

I2C 使能寄存器

表477。IC_ENABLE
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	TX_CMD_BLOCK : 在主机模式下: - 1'b1: 即使 Tx FIFO 中有数据待传输, 也阻止在 I2C 总线上发送数据。- 1'b0: 数据传输在 I2C 总线上自动启动, 当 Tx FIFO 中有第一个数据可用时即刻开始。注意: 仅当 Tx FIFO 为空 (IC_STATUS[2] == 1) 且主机处于空闲状态 (IC_STATUS[5] == 0) 时, 方可设置 TX_CMD_BLOCK 位以阻止主机命令执行。之后放入 Tx FIFO 的任何命令均不会被执行, 直至清除 TX_CMD_BLOCK 位。复位值为: IC_TX_CMD_BLOCK_DEFAULT	读写	0x0
	枚举值:		
	0x0 → NOT_BLOCKED: Tx 命令执行未被阻止		

Bits	Description	Type	Reset
	0x1 → BLOCKED: Tx Command execution blocked		
1	<p>ABORT: When set, the controller initiates the transfer abort. - 0: ABORT not initiated or ABORT done - 1: ABORT operation in progress The software can abort the I2C transfer in master mode by setting this bit. The software can set this bit only when ENABLE is already set; otherwise, the controller ignores any write to ABORT bit. The software cannot clear the ABORT bit once set. In response to an ABORT, the controller issues a STOP and flushes the Tx FIFO after completing the current transfer, then sets the TX_ABORT interrupt after the abort operation. The ABORT bit is cleared automatically after the abort operation.</p> <p>For a detailed description on how to abort I2C transfers, refer to 'Aborting I2C Transfers'.</p> <p>Reset value: 0x0</p>	RW	0x0
	Enumerated values:		
	0x0 → DISABLE: ABORT operation not in progress		
	0x1 → ENABLED: ABORT operation in progress		
0	<p>ENABLE: Controls whether the DW_apb_i2c is enabled. - 0: Disables DW_apb_i2c (TX and RX FIFOs are held in an erased state) - 1: Enables DW_apb_i2c Software can disable DW_apb_i2c while it is active. However, it is important that care be taken to ensure that DW_apb_i2c is disabled properly. A recommended procedure is described in 'Disabling DW_apb_i2c'.</p> <p>When DW_apb_i2c is disabled, the following occurs: - The TX FIFO and RX FIFO get flushed. - Status bits in the IC_INTR_STAT register are still active until DW_apb_i2c goes into IDLE state. If the module is transmitting, it stops as well as deletes the contents of the transmit buffer after the current transfer is complete. If the module is receiving, the DW_apb_i2c stops the current transfer at the end of the current byte and does not acknowledge the transfer.</p> <p>In systems with asynchronous pclk and ic_clk when IC_CLK_TYPE parameter set to asynchronous (1), there is a two ic_clk delay when enabling or disabling the DW_apb_i2c. For a detailed description on how to disable DW_apb_i2c, refer to 'Disabling DW_apb_i2c'</p> <p>Reset value: 0x0</p>	RW	0x0
	Enumerated values:		
	0x0 → DISABLED: I2C is disabled		
	0x1 → ENABLED: I2C is enabled		

I2C: IC_STATUS Register

Offset: 0x70

Description

I2C Status Register

This is a read-only register used to indicate the current transfer status and FIFO status. The status register may be read at any time. None of the bits in this register request an interrupt.

位	描述	类型	复位值
	0x1 → BLOCKED: Tx命令执行已被阻止		
1	<p>ABORT: 设置该位后，控制器启动传输中止操作。- 0：未启动中止或中止已完成； - 1：中止操作正在进行中。软件可通过设置该位在主模式下中止I2C传输。仅当ENABLE位已设置时，软件方可设置该位；否则控制器将忽略对ABORT位的任何写入。一旦设置，软件无法清除该位。响应中止时，控制器将在完成当前传输后发出STOP信号并清空Tx FIFO，随后触发TX_ABORT中断。ABORT 位在中止操作完成后会自动清除。</p> <p>有关如何中止 I2C 传输的详细说明，请参阅“中止 I2C 传输”。</p> <p>复位值：0x0</p>	读写	0x0
	枚举值：		
	0x0 → 禁用：未进行 ABORT 操作		
	0x1 → 启用：正在进行 ABORT 操作		
0	<p>ENABLE: 控制是否启用 DW_apb_i2c。 - 0：禁用 DW_apb_i2c (TX 和 RX FIFO 保持在擦除状态) - 1：启用 DW_apb_i2c。软件可在 DW_apb_i2c 处于活动状态时禁用它。然而，务必谨慎确保 DW_apb_i2c 被正确禁用。推荐的操作步骤详见“禁用 DW_apb_i2c”。</p> <p>当 DW_apb_i2c 被禁用时，发生以下情况： - TX FIFO 和 RX FIFO 会被清空。 - IC_INTR_STAT 寄存器中的状态位仍然有效，直到 DW_apb_i2c 进入空闲 (I DLE) 状态。如果模块正在传输，则在当前传输完成后停止传输并清空传输缓冲区内容。如果模块正在接收，DW_apb_i2c 会在当前字节结束时停止传输且不确认该次传输。</p> <p>在 pclk 与 ic_clk 异步的系统中，当 IC_CLK_TYPE 参数设置为异步 (1) 时，启用或禁用 DW_apb_i2c 会延迟两个 ic_clk 时钟周期。有关如何禁用 DW_apb_i2c 的详细说明，请参见“禁用 DW_apb_i2c”一节。</p> <p>复位值：0x0</p>	读写	0x0
	枚举值：		
	0x0 → DISABLED：I2C 已禁用		
	0x1 → ENABLED：I2C 已启用		

I2C: IC_STATUS 寄存器

偏移：0x70

描述

I2C 状态寄存器

该寄存器为只读，用于指示当前传输状态及 FIFO 状态。状态寄存器可在任意时刻读取。该寄存器中的所有位均未请求中断。

When the I2C is disabled by writing 0 in bit 0 of the IC_ENABLE register: - Bits 1 and 2 are set to 1 - Bits 3 and 10 are set to 0 When the master or slave state machines goes to idle and ic_en=0: - Bits 5 and 6 are set to 0

Table 478. IC_STATUS Register

Bits	Description	Type	Reset
31:7	Reserved.	-	-
6	SLV_ACTIVITY: Slave FSM Activity Status. When the Slave Finite State Machine (FSM) is not in the IDLE state, this bit is set. - 0: Slave FSM is in IDLE state so the Slave part of DW_apb_i2c is not Active - 1: Slave FSM is not in IDLE state so the Slave part of DW_apb_i2c is Active Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → IDLE: Slave is idle		
	0x1 → ACTIVE: Slave not idle		
5	MST_ACTIVITY: Master FSM Activity Status. When the Master Finite State Machine (FSM) is not in the IDLE state, this bit is set. - 0: Master FSM is in IDLE state so the Master part of DW_apb_i2c is not Active - 1: Master FSM is not in IDLE state so the Master part of DW_apb_i2c is Active Note: IC_STATUS[0]-that is, ACTIVITY bit-is the OR of SLV_ACTIVITY and MST_ACTIVITY bits. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → IDLE: Master is idle		
	0x1 → ACTIVE: Master not idle		
4	RFF: Receive FIFO Completely Full. When the receive FIFO is completely full, this bit is set. When the receive FIFO contains one or more empty location, this bit is cleared. - 0: Receive FIFO is not full - 1: Receive FIFO is full Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → NOT_FULL: Rx FIFO not full		
	0x1 → FULL: Rx FIFO is full		
3	RFNE: Receive FIFO Not Empty. This bit is set when the receive FIFO contains one or more entries; it is cleared when the receive FIFO is empty. - 0: Receive FIFO is empty - 1: Receive FIFO is not empty Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → EMPTY: Rx FIFO is empty		
	0x1 → NOT_EMPTY: Rx FIFO not empty		
2	TFE: Transmit FIFO Completely Empty. When the transmit FIFO is completely empty, this bit is set. When it contains one or more valid entries, this bit is cleared. This bit field does not request an interrupt. - 0: Transmit FIFO is not empty - 1: Transmit FIFO is empty Reset value: 0x1	RO	0x1
	Enumerated values:		
	0x0 → NON_EMPTY: Tx FIFO not empty		
	0x1 → EMPTY: Tx FIFO is empty		

当通过向IC_ENABLE寄存器的第0位写入0以禁用I2C时： - 第1位和第2位被置为1 - 第3位和第10位被置为0 当主设备或从设备状态机进入空闲且ic_en=0时： - 第5位和第6位被置为0

表478. IC_STATUS寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6	SLV_ACTIVITY : 从机有限状态机 (FSM) 的活动状态。当从机有限状态机不处于IDLE状态时，该位被置位。- 0: 从机FSM处于IDLE状态，DW_apb_i2c的从机部分处于非活动状态 - 1: 从机FSM不处于IDLE状态，DW_apb_i2c的从机部分处于活动状态。复位值: 0x0 枚举值： 0x0 → IDLE：从机空闲 0x1 → ACTIVE：从机非空闲	只读	0x0
5	MST_ACTIVITY : 主控有限状态机 (FSM) 活动状态。当主控有限状态机不处于IDLE状态时，该位被置位。- 0: 主控FSM处于IDLE状态，DW_apb_i2c的主控部分处于非活动状态 - 1: 主控FSM不处于IDLE状态，DW_apb_i2c的主控部分处于活动状态。注意： IC_STATUS[0]——即ACTIVITY位——是SLV_ACTIVITY和MST_ACTIVITY位的逻辑或。 复位值: 0x0 枚举值： 0x0 → IDLE：主控空闲 0x1 → ACTIVE：主控不空闲	只读	0x0
4	RFF : 接收FIFO完全满。当接收FIFO完全满时，该位被置位；当接收FIFO有一个或多个空位时，该位被清除。- 0: 接收FIFO未满 - 1: 接收FIFO已满。复位值: 0x0 枚举值： 0x0 → NOT_FULL：接收FIFO未满 0x1 → FULL：接收FIFO已满	只读	0x0
3	RFNE : 接收FIFO非空。当接收FIFO中有一条或多条数据时，该位被置位；当接收FIFO为空时，该位被清除。- 0: 接收FIFO为空 - 1: 接收FIFO非空 复位值: 0x0 枚举值： 0x0 → EMPTY：接收FIFO为空 0x1 → NOT_EMPTY：接收FIFO非空	只读	0x0
2	TFE : 发送FIFO完全为空。当发送FIFO完全为空时，该位被置位；只要包含一条或多条有效数据，该位即被清除。该位域不请求中断。- 0: 发送FIFO非空 - 1: 发送FIFO为空 复位值: 0x1 枚举值： 0x0 → NON_EMPTY：发送FIFO非空 0x1 → EMPTY：发送FIFO为空	只读	0x1

Bits	Description	Type	Reset
1	TFNF: Transmit FIFO Not Full. Set when the transmit FIFO contains one or more empty locations, and is cleared when the FIFO is full. - 0: Transmit FIFO is full - 1: Transmit FIFO is not full Reset value: 0x1	RO	0x1
	Enumerated values:		
	0x0 → FULL: Tx FIFO is full		
	0x1 → NOT_FULL: Tx FIFO not full		
0	ACTIVITY: I2C Activity Status. Reset value: 0x0	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: I2C is idle		
	0x1 → ACTIVE: I2C is active		

I2C: IC_TXFLR Register

Offset: 0x74

Description

I2C Transmit FIFO Level Register This register contains the number of valid data entries in the transmit FIFO buffer. It is cleared whenever: - The I2C is disabled - There is a transmit abort - that is, TX_ABRT bit is set in the IC_RAW_INTR_STAT register - The slave bulk transmit mode is aborted The register increments whenever data is placed into the transmit FIFO and decrements when data is taken from the transmit FIFO.

Table 479. IC_TXFLR Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	TXFLR: Transmit FIFO Level. Contains the number of valid data entries in the transmit FIFO. Reset value: 0x0	RO	0x00

I2C: IC_RXFLR Register

Offset: 0x78

Description

I2C Receive FIFO Level Register This register contains the number of valid data entries in the receive FIFO buffer. It is cleared whenever: - The I2C is disabled - Whenever there is a transmit abort caused by any of the events tracked in IC_TX_ABRT_SOURCE The register increments whenever data is placed into the receive FIFO and decrements when data is taken from the receive FIFO.

Table 480. IC_RXFLR Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4:0	RXFLR: Receive FIFO Level. Contains the number of valid data entries in the receive FIFO. Reset value: 0x0	RO	0x00

I2C: IC_SDA_HOLD Register

Offset: 0x7c

位	描述	类型	复位值
1	TFNF : 发送FIFO未满。当发送FIFO有一个或多个空位时，该位被置位；当FIFO已满时，该位被清除。- 0: 发送FIFO已满 - 1: 发送FIFO未满 复位值: 0x1	只读	0x1
	枚举值:		
	0x0 → FULL: 发送FIFO已满		
	0x1 → NOT_FULL: 发送FIFO未满		
0	ACTIVITY : I2C活动状态。复位值: 0x0	只读	0x0
	枚举值:		
	0x0 → INACTIVE: I2C处于空闲状态		
	0x1 → ACTIVE: I2C处于活动状态		

I2C: IC_TXFLR寄存器

偏移: 0x74

描述

I2C发送FIFO级别寄存器 该寄存器包含发送FIFO缓冲区中有效数据项的数量。

当满足以下条件时清除: - 禁用 I2C - 发生传输中止, 即 IC_RAW_INTR_STAT 寄存器中的 TX_ABRT 位被置位 - 从机批量传输模式中止。每当数据被放入传输 FIFO 时, 寄存器递增; 当数据从传输 FIFO 取出时, 寄存器递减。

表 479. IC_TXFLR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	TXFLR : 发送FIFO级别。表示发送FIFO中有效数据条目的数量。 复位值: 0x0	只读	0x00

I2C: IC_RXFLR寄存器

偏移: 0x78

描述

I2C接收FIFO级别寄存器。该寄存器表示接收FIFO缓冲区中有效数据条目的数量。该寄存器在以下情况下清零: - I2C 被禁用 - 由IC_TX_ABRT_SOURCE中跟踪的任一事件导致的发送中止。每当数据写入接收FIFO, 寄存器值递增; 每当数据从接收FIFO取出, 寄存器值递减。

表 480. IC_RXFLR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4:0	RXFLR : 接收FIFO级别。表示接收FIFO中有效数据条目的数量。 复位值: 0x0	只读	0x00

I2C: IC_SDA_HOLD寄存器

偏移: 0x7c

Description

I2C SDA Hold Time Length Register

The bits [15:0] of this register are used to control the hold time of SDA during transmit in both slave and master mode (after SCL goes from HIGH to LOW).

The bits [23:16] of this register are used to extend the SDA transition (if any) whenever SCL is HIGH in the receiver in either master or slave mode.

Writes to this register succeed only when IC_ENABLE[0]=0.

The values in this register are in units of ic_clk period. The value programmed in IC_SDA_TX_HOLD must be greater than the minimum hold time in each mode (one cycle in master mode, seven cycles in slave mode) for the value to be implemented.

The programmed SDA hold time during transmit (IC_SDA_TX_HOLD) cannot exceed at any time the duration of the low part of scl. Therefore the programmed value cannot be larger than N_SCL_LOW-2, where N_SCL_LOW is the duration of the low part of the scl period measured in ic_clk cycles.

*Table 481.
IC_SDA_HOLD
Register*

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:16	IC_SDA_RX_HOLD: Sets the required SDA hold time in units of ic_clk period, when DW_apb_i2c acts as a receiver. Reset value: IC_DEFAULT_SDA_HOLD[23:16].	RW	0x00
15:0	IC_SDA_TX_HOLD: Sets the required SDA hold time in units of ic_clk period, when DW_apb_i2c acts as a transmitter. Reset value: IC_DEFAULT_SDA_HOLD[15:0].	RW	0x0001

I2C: IC_TX_ABRT_SOURCE Register

Offset: 0x80

Description

I2C Transmit Abort Source Register

This register has 32 bits that indicate the source of the TX_ABRT bit. Except for Bit 9, this register is cleared whenever the IC_CLR_TX_ABRT register or the IC_CLR_INTR register is read. To clear Bit 9, the source of the ABRT_SBYTE_NORSTRT must be fixed first; RESTART must be enabled (IC_CON[5]=1), the SPECIAL bit must be cleared (IC_TAR[11]), or the GC_OR_START bit must be cleared (IC_TAR[10]).

Once the source of the ABRT_SBYTE_NORSTRT is fixed, then this bit can be cleared in the same manner as other bits in this register. If the source of the ABRT_SBYTE_NORSTRT is not fixed before attempting to clear this bit, Bit 9 clears for one cycle and is then re-asserted.

*Table 482.
IC_TX_ABRT_SOURCE
Register*

Bits	Description	Type	Reset
31:23	TX_FLUSH_CNT: This field indicates the number of Tx FIFO Data Commands which are flushed due to TX_ABRT interrupt. It is cleared whenever I2C is disabled. Reset value: 0x00 Role of DW_apb_i2c: Master-Transmitter or Slave-Transmitter	RO	0x000
22:17	Reserved.	-	-

描述

I2C SDA 保持时间寄存器

该寄存器的[15:0]位用于控制在主从模式下传输过程中SDA信号的保持时间
(在SCL由高电平变为低电平后)。

该寄存器的[23:16]位用于在主从模式的接收端，当SCL为高电平时，延长SDA信号的过渡时间（如有）。

仅当 IC_ENABLE[0]=0 时，写入该寄存器操作方可成功。

该寄存器中的数值以 ic_clk 周期为单位。IC_SDA_RX_HOLD 中编程的值必须大于各模式下的最小保持时间（主模式为一个周期，从模式为七个周期），该值方能生效。

传输期间编程的 SDA 保持时间 (IC_SDA_TX_HOLD) 任意时刻均不得超过 scl 低电平的持续时间。因此，编程值不得大于 N_SCL_LOW-2，其中 N_SCL_LOW 为以 ic_clk 周期计量的 scl 低电平持续时间。

表 481。
IC_SDA_HOLD
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	IC_SDA_RX_HOLD : 设置当 DW_apb_i2c 作为接收器时所需的 SDA 保持时间，单位为 ic_clk 周期。 复位值：IC_DEFAULT_SDA_HOLD[23:16]。	读写	0x00
15:0	IC_SDA_TX_HOLD : 设置当 DW_apb_i2c 作为发送器时所需的 SDA 保持时间，单位为 ic_clk 周期。 复位值：IC_DEFAULT_SDA_HOLD[15:0]。	读写	0x0001

I2C: IC_TX_ABRT_SOURCE 寄存器

偏移: 0x80

描述

I2C 发送中止源寄存器

该寄存器包含32位，用于指示TX_ABRT位的来源。除第9位外，每当读取IC_CLR_TX_ABRT寄存器或IC_CLR_INTR寄存器时，该寄存器即被清除。要清除第9位，必须先修正ABRT_SBYTE_NORSTRT的来源；必须启用RESTART (IC_CON[5]=1)，且SPECIAL位 (IC_TAR[11]) 必须被清除，或者GC_OR_START位 (IC_TAR[10]) 必须被清除。

一旦修正了ABRT_SBYTE_NORSTRT的来源，即可用本寄存器其他位相同的方式清除此位。若在尝试清除此位之前未修正 ABRT_SBYTE_NORSTRT 的来源，第9位将仅清除一个周期，随后再次被置位。

表 482。
IC_TX_ABRT_SOURCE
寄存器

位	描述	类型	复位值
31:23	TX_FLUSH_CNT : 此字段指示因 TX_ABRT 中断而被清空的 Tx FIFO 数据命令数量。当 I2C 被禁用时，该字段将被清零。 复位值：0x0 DW_apb_i2c 角色：主机发送器或从机发送器	只读	0x000
22:17	保留。	-	-

Bits	Description	Type	Reset
16	<p>ABRT_USER_ABRT: This is a master-mode-only bit. Master has detected the transfer abort ([IC_ENABLE[1]])</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_USER_ABRT_VOID: Transfer abort detected by master- scenario not present		
	0x1 → ABRT_USER_ABRT_GENERATED: Transfer abort detected by master		
15	<p>ABRT_SLVRD_INTX: 1: When the processor side responds to a slave mode request for data to be transmitted to a remote master and user writes a 1 in CMD (bit 8) of IC_DATA_CMD register.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Slave-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_SLVRD_INTX_VOID: Slave trying to transmit to remote master in read mode- scenario not present		
	0x1 → ABRT_SLVRD_INTX_GENERATED: Slave trying to transmit to remote master in read mode		
14	<p>ABRT_SLV_ARBLOST: This field indicates that a Slave has lost the bus while transmitting data to a remote master. IC_TX_ABRT_SOURCE[12] is set at the same time. Note: Even though the slave never 'owns' the bus, something could go wrong on the bus. This is a fail safe check. For instance, during a data transmission at the low-to-high transition of SCL, if what is on the data bus is not what is supposed to be transmitted, then DW_apb_i2c no longer own the bus.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Slave-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_SLV_ARBLOST_VOID: Slave lost arbitration to remote master- scenario not present		
	0x1 → ABRT_SLV_ARBLOST_GENERATED: Slave lost arbitration to remote master		
13	<p>ABRT_SLVFLUSH_TXFIFO: This field specifies that the Slave has received a read command and some data exists in the TX FIFO, so the slave issues a TX_ABRT interrupt to flush old data in TX FIFO.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Slave-Transmitter</p>	RO	0x0
	Enumerated values:		

位	描述	类型	复位值
16	<p>ABRT_USER_ABRT: 该位仅适用于主模式。主机检测到传输中止 (IC_ENABLE[1])</p> <p>复位值: 0x0</p> <p>DW_apb_i2c 角色: 主机发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_USER_ABRT_VOID: 主机检测到的传输中止场景不存在		
	0x1 → ABRT_USER_ABRT_GENERATED: 主机检测到的传输中止		
15	<p>ABRT_SLVRD_INTX: 1: 当处理器侧响应从模式请求, 向远程主机传输数据, 且用户在 IC_DATA_CMD 寄存器的 CMD (第8位) 写入1时。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SLVRD_INTX_VOID: 从机尝试在读取模式下向远程主机发送数据——此情景不存在		
	0x1 → ABRT_SLVRD_INTX_GENERATED: 从机尝试在读取模式下向远程主机发送数据		
14	<p>ABRT_SLV_ARBLOST: 该字段指示从机在向远程主机传输数据时丢失总线控制权。同时设置了IC_TX_ABRT_SOURCE[12]。注意: 尽管从机从未“拥有”总线, 但总线上仍可能发生故障。本项为故障保护检测。例如, 在SCL由低到高的传输过程中, 如果数据总线上的数据与应传输的数据不符, 则DW_apb_i2c即不再拥有该总线。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SLV_ARBLOST_VOID: 从机在仲裁中败给远程主机——方案不存在		
	0x1 → ABRT_SLV_ARBLOST_GENERATED: 从机在仲裁中败给远程主设备		
13	<p>ABRT_SLVFLUSH_TXFIFO: 该字段表示从机已接收到读取命令且TX FIFO中存在数据, 因此从机发出TX_ABRT中断以清除TX FIFO中的旧数据。</p> <p>复位值: 0x0</p> <p>DW_apb_i2c的角色: 从机-发送器</p>	只读	0x0
	枚举值:		

Bits	Description	Type	Reset
	0x0 → ABRT_SLVFLUSH_TXFIFO_VOID: Slave flushes existing data in TX-FIFO upon getting read command- scenario not present		
	0x1 → ABRT_SLVFLUSH_TXFIFO_GENERATED: Slave flushes existing data in TX-FIFO upon getting read command		
12	<p>ARB_LOST: This field specifies that the Master has lost arbitration, or if IC_TX_ABRT_SOURCE[14] is also set, then the slave transmitter has lost arbitration.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter or Slave-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_LOST_VOID: Master or Slave-Transmitter lost arbitration- scenario not present		
	0x1 → ABRT_LOST_GENERATED: Master or Slave-Transmitter lost arbitration		
11	<p>ABRT_MASTER_DIS: This field indicates that the User tries to initiate a Master operation with the Master mode disabled.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter or Master-Receiver</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_MASTER_DIS_VOID: User initiating master operation when MASTER disabled- scenario not present		
	0x1 → ABRT_MASTER_DIS_GENERATED: User initiating master operation when MASTER disabled		
10	<p>ABRT_10B_RD_NORSTRT: This field indicates that the restart is disabled (IC_RESTART_EN bit (IC_CON[5]) =0) and the master sends a read command in 10-bit addressing mode.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Receiver</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_10B_RD_VOID: Master not trying to read in 10Bit addressing mode when RESTART disabled		
	0x1 → ABRT_10B_RD_GENERATED: Master trying to read in 10Bit addressing mode when RESTART disabled		

位	描述	类型	复位值
	0x0 → ABRT_SLVFLUSH_TXFIFO_VOID：从机在收到读取命令时清除TX FIFO 中现有数据——方案不存在		
	0x1 → ABRT_SLVFLUSH_TXFIFO_GENERATED：从机在收到读取命令时清除 TX FIFO 中现有数据		
12	<p>ARB_LOST：该字段表示主设备失去仲裁，或在IC_TX_ABRT_SOURCE[1:4]置位时表示从机发送器失去仲裁。</p> <p>复位值：0x0</p> <p>DW_apb_i2c 角色：主机发送器或从机发送器</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_LOST_VOID：主设备或从机发送器失去仲裁——方案不存在		
	0x1 → ABRT_LOST_GENERATED：主机或从机发送端仲裁丢失		
11	<p>ABRT_MASTER_DIS：该字段指示用户在主机模式禁用时尝试启动主机操作。</p> <p>复位值：0x0</p> <p>DW_apb_i2c的角色：主机发送端或主机接收端</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_MASTER_DIS_VOID：用户在主机禁用时启动主机操作的情形不存在		
	0x1 → ABRT_MASTER_DIS_GENERATED：用户在主机禁用时启动主机操作		
10	<p>ABRT_10B_RD_NORSTRT：该字段表示重启功能被禁用 (IC_RESTART_EN位 (IC_CON[5]) =0) ，且主机在10位地址模式下发送读取命令。</p> <p>复位值：0x0</p> <p>DW_apb_i2c的角色：主机接收端</p>	只读	0x0
	枚举值：		
	0x0 → ABRT_10B_RD_VOID：当重启功能禁用时，主机未尝试在10位地址模式下读取		
	0x1 → ABRT_10B_RD_GENERATED：主控设备在RESTART禁用时尝试以10位寻址模式读取		

Bits	Description	Type	Reset
9	ABRT_SBYTE_NORSTRT: To clear Bit 9, the source of the ABRT_SBYTE_NORSTRT must be fixed first; restart must be enabled (IC_CON[5]=1), the SPECIAL bit must be cleared (IC_TAR[11]), or the GC_OR_START bit must be cleared (IC_TAR[10]). Once the source of the ABRT_SBYTE_NORSTRT is fixed, then this bit can be cleared in the same manner as other bits in this register. If the source of the ABRT_SBYTE_NORSTRT is not fixed before attempting to clear this bit, bit 9 clears for one cycle and then gets reasserted. When this field is set to 1, the restart is disabled (IC_RESTART_EN bit (IC_CON[5]) =0) and the user is trying to send a START Byte. Reset value: 0x0 Role of DW_apb_i2c: Master	RO	0x0
	Enumerated values:		
	0x0 → ABRT_SBYTE_NORSTRT_VOID: User trying to send START byte when RESTART disabled- scenario not present		
	0x1 → ABRT_SBYTE_NORSTRT_GENERATED: User trying to send START byte when RESTART disabled		
8	ABRT_HS_NORSTRT: This field indicates that the restart is disabled (IC_RESTART_EN bit (IC_CON[5]) =0) and the user is trying to use the master to transfer data in High Speed mode. Reset value: 0x0 Role of DW_apb_i2c: Master-Transmitter or Master-Receiver	RO	0x0
	Enumerated values:		
	0x0 → ABRT_HS_NORSTRT_VOID: User trying to switch Master to HS mode when RESTART disabled- scenario not present		
	0x1 → ABRT_HS_NORSTRT_GENERATED: User trying to switch Master to HS mode when RESTART disabled		
7	ABRT_SBYTE_ACKDET: This field indicates that the Master has sent a START Byte and the START Byte was acknowledged (wrong behavior). Reset value: 0x0 Role of DW_apb_i2c: Master	RO	0x0
	Enumerated values:		
	0x0 → ABRT_SBYTE_ACKDET_VOID: ACK detected for START byte- scenario not present		
	0x1 → ABRT_SBYTE_ACKDET_GENERATED: ACK detected for START byte		
6	ABRT_HS_ACKDET: This field indicates that the Master is in High Speed mode and the High Speed Master code was acknowledged (wrong behavior). Reset value: 0x0 Role of DW_apb_i2c: Master	RO	0x0

位	描述	类型	复位值
9	<p>ABRT_SBYTE_NORSTRT: 要清除第9位，必须先修正ABRT_SBYTE_NORSTRT的根本原因；必须启用restart (IC_CON[5]=1)，清除SPECIAL位 (IC_TAR[11])，或清除GC_OR_START位 (IC_TAR[10])。一旦修正ABRT_SBYTE_NORSTRT的根本原因，该位即可按此寄存器中其他位的相同方式清除。若未修正ABRT_SBYTE_NORSTRT的根本原因而试图清除此位，则第9位会被清除一个周期后重新置位。当该字段设置为1时，restart被禁用 (IC_RESTART_EN位(IC_CON[5])=0)，且用户正在尝试发送START字节。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SBYTE_NORSTRT_VOID : 用户尝试在禁用重启 (RESTART) 时发送 START 字节——该场景不存在		
	0x1 → ABRT_SBYTE_NORSTRT_GENERATED : 用户尝试在禁用重启 (RESTART) 时发送 START 字节		
8	<p>ABRT_HS_NORSTRT: 该字段指示重启功能被禁用 (IC_RESTART_EN 位 (IC_CON[5]) =0) 且用户尝试使用主控以高速模式传输数据。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主机发送端或主机接收端</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_HS_NORSTRT_VOID : 用户尝试在禁用重启 (RESTART) 时切换主控至高速模式——该场景不存在		
	0x1 → ABRT_HS_NORSTRT_GENERATED : 用户尝试在禁用重启 (RESTART) 时切换主控至高速模式		
7	<p>ABRT_SBYTE_ACKDET: 该字段指示主控已发送 START 字节且该 START 字节被应答（错误行为）。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0
	枚举值:		
	0x0 → ABRT_SBYTE_ACKDET_VOID : 检测到START字节的ACK——该情况不存在		
	0x1 → ABRT_SBYTE_ACKDET_GENERATED : 检测到START字节的ACK		
6	<p>ABRT_HS_ACKDET: 该字段表示主机处于高速模式且高速主机代码已被确认（为不当行为）。</p> <p>复位值: 0x0 DW_apb_i2c 的角色: 主控</p>	只读	0x0

Bits	Description	Type	Reset
	Enumerated values:		
	0x0 → ABRT_HS_ACK_VOID: HS Master code ACKed in HS Mode- scenario not present		
	0x1 → ABRT_HS_ACK_GENERATED: HS Master code ACKed in HS Mode		
5	<p>ABRT_GCALL_READ: This field indicates that DW_apb_i2c in the master mode has sent a General Call but the user programmed the byte following the General Call to be a read from the bus (IC_DATA_CMD[9] is set to 1).</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_GCALL_READ_VOID: GCALL is followed by read from bus-scenario not present		
	0x1 → ABRT_GCALL_READ_GENERATED: GCALL is followed by read from bus		
4	<p>ABRT_GCALL_NOACK: This field indicates that DW_apb_i2c in master mode has sent a General Call and no slave on the bus acknowledged the General Call.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_GCALL_NOACK_VOID: GCALL not ACKed by any slave-scenario not present		
	0x1 → ABRT_GCALL_NOACK_GENERATED: GCALL not ACKed by any slave		
3	<p>ABRT_TXDATA_NOACK: This field indicates the master-mode only bit. When the master receives an acknowledgement for the address, but when it sends data byte(s) following the address, it did not receive an acknowledge from the remote slave(s).</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter</p>	RO	0x0
	Enumerated values:		
	0x0 → ABRT_TXDATA_NOACK_VOID: Transmitted data non-ACKed by addressed slave-scenario not present		
	0x1 → ABRT_TXDATA_NOACK_GENERATED: Transmitted data not ACKed by addressed slave		
2	<p>ABRT_10ADDR2_NOACK: This field indicates that the Master is in 10-bit address mode and that the second address byte of the 10-bit address was not acknowledged by any slave.</p> <p>Reset value: 0x0</p> <p>Role of DW_apb_i2c: Master-Transmitter or Master-Receiver</p>	RO	0x0

位	描述	类型	复位值
	枚举值：		
	0x0 → ABRT_HS_ACK_VOID：高速模式下高速主机代码被确认——该情况不存在		
	0x1 → ABRT_HS_ACK_GENERATED：高速模式下高速主机代码被确认		
5	ABRT_GCALL_READ : 该字段表示DW_apb_i2c处于主机模式时发送了广播呼叫，但用户将广播呼叫后的字节编程为从总线读取 (IC_DATA_CMD[9] 设置为1)。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_GCALL_READ_VOID：广播呼叫后接从总线读取——该情况不存在		
	0x1 → ABRT_GCALL_READ_GENERATED：GCALL 后紧跟对总线的读取		
4	ABRT_GCALL_NOACK : 该字段指示 DW_apb_i2c 主模式已发送 General Call，但总线上的任何从设备均未确认该 General Call。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_GCALL_NOACK_VOID：未有任何从设备确认 GCALL 的情况不存在		
	0x1 → ABRT_GCALL_NOACK_GENERATED：未有任何从设备确认 GCALL		
3	ABRT_TXDATA_NOACK : 该字段为仅限主模式位。当主设备接收到地址确认后，发送后续数据字节时未收到远程从设备确认。 复位值: 0x0 DW_apb_i2c 角色: 主机发送器	只读	0x0
	枚举值：		
	0x0 → ABRT_TXDATA_NOACK_VOID：发送的数据未被指定从设备确认的情况不存在		
	0x1 → ABRT_TXDATA_NOACK_GENERATED：发送的数据未被指定从设备确认		
2	ABRT_10ADDR2_NOACK : 该字段指示主机处于10位地址模式，且10位地址的第二字节未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色: 主机发送端或主机接收端	只读	0x0

Bits	Description	Type	Reset
	Enumerated values:		
	0x0 → INACTIVE: This abort is not generated		
	0x1 → ACTIVE: Byte 2 of 10Bit Address not ACKed by any slave		
1	ABRT_10ADDR1_NOACK: This field indicates that the Master is in 10-bit address mode and the first 10-bit address byte was not acknowledged by any slave. Reset value: 0x0 Role of DW_apb_i2c: Master-Transmitter or Master-Receiver	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: This abort is not generated		
	0x1 → ACTIVE: Byte 1 of 10Bit Address not ACKed by any slave		
0	ABRT_7B_ADDR_NOACK: This field indicates that the Master is in 7-bit addressing mode and the address sent was not acknowledged by any slave. Reset value: 0x0 Role of DW_apb_i2c: Master-Transmitter or Master-Receiver	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: This abort is not generated		
	0x1 → ACTIVE: This abort is generated because of NOACK for 7-bit address		

I2C: IC_SLV_DATA_NACK_ONLY Register

Offset: 0x84

Description

Generate Slave Data NACK Register

The register is used to generate a NACK for the data part of a transfer when DW_apb_i2c is acting as a slave-receiver. This register only exists when the IC_SLV_DATA_NACK_ONLY parameter is set to 1. When this parameter disabled, this register does not exist and writing to the register's address has no effect.

A write can occur on this register if both of the following conditions are met: - DW_apb_i2c is disabled (IC_ENABLE[0] = 0) - Slave part is inactive (IC_STATUS[6] = 0) Note: The IC_STATUS[6] is a register read-back location for the internal slv_activity signal; the user should poll this before writing the ic_slv_data_nack_only bit.

Table 483.
IC_SLV_DATA_NACK_ONLY Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	NACK: Generate NACK. This NACK generation only occurs when DW_apb_i2c is a slave-receiver. If this register is set to a value of 1, it can only generate a NACK after a data byte is received; hence, the data transfer is aborted and the data received is not pushed to the receive buffer. When the register is set to a value of 0, it generates NACK/ACK, depending on normal criteria. - 1: generate NACK after data byte received - 0: generate NACK/ACK normally Reset value: 0x0	RW	0x0
	Enumerated values:		

位	描述	类型	复位值
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：10位地址的第2字节未被任何从机确认		
1	ABRT_10ADDR1_NOACK : 该字段指示主机处于10位地址模式，且第一字节10位地址未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色：主机发送端或主机接收端	只读	0x0
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：10位地址的第1字节未被任何从机确认		
0	ABRT_7B_ADDR_NOACK : 该字段指示主机处于7位地址模式，且发送的地址未被任何从机确认。 复位值: 0x0 DW_apb_i2c的角色：主机发送端或主机接收端	只读	0x0
	枚举值： 0x0 → 非活动：未生成此中止信号 0x1 → 活动：因7位地址未响应（NOACK）而生成此中止信号		

I2C: IC_SLV_DATA_NACK_ONLY 寄存器

偏移: 0x84

描述

生成从设备数据 NACK 寄存器

该寄存器用于在DW_apb_i2c作为从机接收器时，为传输的数据段生成NACK。

仅当IC_SLV_DATA_NACK_ONLY参数设置为1时，该寄存器才存在。当该参数被禁用时，该寄存器不存在，写入该寄存器地址不会产生任何效果。

仅当满足以下两个条件时，才能对该寄存器进行写入： - DW_apb_i2c 被禁用 (IC_ENABLE[0] = 0) - 从机部分处于非活动状态 (IC_STATUS[6] = 0)
注：IC_STATUS[6] 是内部 slv_activity 信号的寄存器读取反馈位；用户应在写入 ic_slv_data_nack_only 位之前，先轮询该位。

表 483。
IC_SLV_DATA_NACK_ONLY 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	NACK : 生成 NACK。该 NACK 仅在 DW_apb_i2c 作为从机接收器时生成。如果该寄存器设置为1，则仅在接收到数据字节后生成NACK；因此，数据传输被中止，接收的数据不会被推入接收缓冲区。 当寄存器设置为0时，根据正常条件生成NACK或ACK。- 1: 接收数据字节后生成NACK - 0: 正常生成NACK或ACK 重置值: 0x0	读写	0x0
	枚举值：		

Bits	Description	Type	Reset
	0x0 → DISABLED: Slave receiver generates NACK normally		
	0x1 → ENABLED: Slave receiver generates NACK upon data reception only		

I2C: IC_DMA_CR Register

Offset: 0x88

Description

DMA Control Register

The register is used to enable the DMA Controller interface operation. There is a separate bit for transmit and receive. This can be programmed regardless of the state of IC_ENABLE.

Table 484.
IC_DMA_CR Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	TDMAE: Transmit DMA Enable. This bit enables/disables the transmit FIFO DMA channel. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → DISABLED: transmit FIFO DMA channel disabled		
	0x1 → ENABLED: Transmit FIFO DMA channel enabled		
0	RDMAE: Receive DMA Enable. This bit enables/disables the receive FIFO DMA channel. Reset value: 0x0	RW	0x0
	Enumerated values:		
	0x0 → DISABLED: Receive FIFO DMA channel disabled		
	0x1 → ENABLED: Receive FIFO DMA channel enabled		

I2C: IC_DMA_TDLR Register

Offset: 0x8c

Description

DMA Transmit Data Level Register

Table 485.
IC_DMA_TDLR Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3:0	DMATDL: Transmit Data Level. This bit field controls the level at which a DMA request is made by the transmit logic. It is equal to the watermark level; that is, the dma_tx_req signal is generated when the number of valid data entries in the transmit FIFO is equal to or below this field value, and TDMAE = 1. Reset value: 0x0	RW	0x0

I2C: IC_DMA_RDLR Register

Offset: 0x90

Description

I2C Receive Data Level Register

位	描述	类型	复位值
	0x0 → 禁用：从机接收器正常生成NACK		
	0x1 → 启用：从机接收器仅在接收数据时生成NACK		

I2C: IC_DMA_CR 寄存器

偏移量: 0x88

描述

DMA 控制寄存器

该寄存器用于使能 DMA 控制器接口操作。发送和接收各自拥有独立控制位。

可在 IC_ENABLE 状态下任意编程。

表 484。
IC_DMA_CR 寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TDMAE : 传输DMA使能。此位用于启用或禁用传输FIFO DMA通道。复位值： : 0x0	读写	0x0
	枚举值：		
	0x0 → 禁用：传输FIFO DMA通道被禁用		
	0x1 → 启用：传输FIFO DMA通道被启用		
0	RDMAE : 接收DMA使能。此位用于启用或禁用接收FIFO DMA通道。复位值： 0x0	读写	0x0
	枚举值：		
	0x0 → 禁用：接收FIFO DMA通道被禁用		
	0x1 → 启用：接收FIFO DMA通道被启用		

I2C: IC_DMA_TDRL 寄存器

偏移量: 0x8c

说明

DMA 传输数据级别寄存器

表 485。
IC_DMA_TDRL
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	DMATDL : 传输数据级别。该字段控制传输逻辑触发DMA请求的阈值。其值等同于水位线；即当传输FIFO中有效数据条目数小于或等于该字段值且 TDMAE=1 时，dma_tx_req信号被激活。 复位值：0x0	读写	0x0

I2C: IC_DMA_RDRL 寄存器

偏移：0x90

描述

I2C 接收数据水位寄存器

Table 486.
IC_DMA_RDLR
Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3:0	DMARDL: Receive Data Level. This bit field controls the level at which a DMA request is made by the receive logic. The watermark level = DMARDL+1; that is, dma_rx_req is generated when the number of valid data entries in the receive FIFO is equal to or more than this field value + 1, and RDMAE =1. For instance, when DMARDL is 0, then dma_rx_req is asserted when 1 or more data entries are present in the receive FIFO. Reset value: 0x0	RW	0x0

I2C: IC_SDA_SETUP Register

Offset: 0x94

Description

I2C SDA Setup Register

This register controls the amount of time delay (in terms of number of ic_clk clock periods) introduced in the rising edge of SCL - relative to SDA changing - when DW_apb_i2c services a read request in a slave-transmitter operation. The relevant I2C requirement is tSU:DAT (note 4) as detailed in the I2C Bus Specification. This register must be programmed with a value equal to or greater than 2.

Writes to this register succeed only when IC_ENABLE[0] = 0.

Note: The length of setup time is calculated using [(IC_SDA_SETUP - 1) * (ic_clk_period)], so if the user requires 10 ic_clk periods of setup time, they should program a value of 11. The IC_SDA_SETUP register is only used by the DW_apb_i2c when operating as a slave transmitter.

Table 487.
IC_SDA_SETUP
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	SDA_SETUP: SDA Setup. It is recommended that if the required delay is 1000ns, then for an ic_clk frequency of 10 MHz, IC_SDA_SETUP should be programmed to a value of 11. IC_SDA_SETUP must be programmed with a minimum value of 2.	RW	0x64

I2C: IC_ACK_GENERAL_CALL Register

Offset: 0x98

Description

I2C ACK General Call Register

The register controls whether DW_apb_i2c responds with a ACK or NACK when it receives an I2C General Call address.

This register is applicable only when the DW_apb_i2c is in slave mode.

Table 488.
IC_ACK_GENERAL_CA
LL Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	ACK_GEN_CALL: ACK General Call. When set to 1, DW_apb_i2c responds with a ACK (by asserting ic_data_oe) when it receives a General Call. Otherwise, DW_apb_i2c responds with a NACK (by negating ic_data_oe).	RW	0x1
	Enumerated values:		
	0x0 → DISABLED: Generate NACK for a General Call		
	0x1 → ENABLED: Generate ACK for a General Call		

表 486。
IC_DMA_RDLR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	DMARDL : 接收数据水位。此字段控制接收逻辑发送DMA请求的水位。水位线 = DMARDL + 1; 即当接收FIFO中有效数据条目数大于或等于该字段值加1且RDMAE = 1时, dma_rx_req信号被触发。例如, 当DMARDL为0时, 接收FIFO中存在1条或以上数据时, 将触发dma_rx_req。 复位值: 0x0	读写	0x0

I2C: IC_SDA_SETUP 寄存器

偏移: 0x94

描述

I2C SDA 设置寄存器

本寄存器控制DW_apb_i2c在从机作为发射端响应读请求时, SCL上升沿相对于SDA变化所引入的延迟时间（以ic_clk时钟周期计）。相关的I2C要求为tSU:DAT（注4），详见I2C总线规格。本寄存器必须编程为大于或等于2的值。

仅当IC_ENABLE[0] = 0时, 写入此寄存器操作才会成功。

注：设置时间的长度通过[(IC_SDA_SETUP - 1) * (ic_clk_period)]计算，因此若用户需要10个ic_clk周期的设置时间，应编程为11。IC_SDA_SETUP寄存器仅在DW_apb_i2c作为从机发送时使用。

表 487。
IC_SDA_SETUP
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SDA_SETUP : SDA设置。建议当所需延迟为1000纳秒且ic_clk频率为10 MHz时, IC_SDA_SETUP应编程为11。IC_SDA_SETUP寄存器的最小编程值为2。	读写	0x64

I2C: IC_ACK_GENERAL_CALL 寄存器

偏移: 0x98

说明

I2C 应答通用调用寄存器

该寄存器用于控制 DW_apb_i2c 接收到 I2C 通用呼叫地址时, 响应 ACK 还是 NACK。

该寄存器仅在 DW_apb_i2c 处于从模式时适用。

表 488。
IC_ACK_GENERAL_CA
LL 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	ACK_GEN_CALL : ACK 通用呼叫。设置为 1 时, DW_apb_i2c 在接收到通用呼叫时通过置位 ic_data_oe 响应 ACK; 否则, 通过清除 ic_data_oe 响应 NACK。 枚举值: 0x0 → 禁用: 对通用呼叫生成 NACK 0x1 → 启用: 对通用呼叫生成 ACK	读写	0x1

I2C: IC_ENABLE_STATUS Register

Offset: 0x9c

Description

I2C Enable Status Register

The register is used to report the DW_apb_i2c hardware status when the IC_ENABLE[0] register is set from 1 to 0; that is, when DW_apb_i2c is disabled.

If IC_ENABLE[0] has been set to 1, bits 2:1 are forced to 0, and bit 0 is forced to 1.

If IC_ENABLE[0] has been set to 0, bits 2:1 is only valid as soon as bit 0 is read as '0'.

Note: When IC_ENABLE[0] has been set to 0, a delay occurs for bit 0 to be read as 0 because disabling the DW_apb_i2c depends on I2C bus activities.

Table 489.
IC_ENABLE_STATUS
Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	<p>SLV_RX_DATA_LOST: Slave Received Data Lost. This bit indicates if a Slave-Receiver operation has been aborted with at least one data byte received from an I2C transfer due to the setting bit 0 of IC_ENABLE from 1 to 0. When read as 1, DW_apb_i2c is deemed to have been actively engaged in an aborted I2C transfer (with matching address) and the data phase of the I2C transfer has been entered, even though a data byte has been responded with a NACK.</p> <p>Note: If the remote I2C master terminates the transfer with a STOP condition before the DW_apb_i2c has a chance to NACK a transfer, and IC_ENABLE[0] has been set to 0, then this bit is also set to 1.</p> <p>When read as 0, DW_apb_i2c is deemed to have been disabled without being actively involved in the data phase of a Slave-Receiver transfer.</p> <p>Note: The CPU can safely read this bit when IC_EN (bit 0) is read as 0.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: Slave RX Data is not lost		
	0x1 → ACTIVE: Slave RX Data is lost		

I2C: IC_ENABLE_STATUS 寄存器

偏移: 0x9c

描述

I2C 使能状态寄存器

当 IC_ENABLE[0] 寄存器由 1 变更为 0 时，该寄存器用于报告 DW_apb_i2c 硬件状态；即，当 DW_apb_i2c 被禁用时。

如果 IC_ENABLE[0] 被设置为 1，则位 2:1 被强制设为 0，且位 0 被强制设为 1。

如果 IC_ENABLE[0] 被设置为 0，则仅当位 0 读取为“0”时，位 2:1 才有效。

注意：当 IC_ENABLE[0] 被设置为 0 时，位 0 读取为 0 会出现延迟，因为禁用 DW_apb_i2c 依赖于 I2C 总线活动。

表 489。
IC_ENABLE_STATUS
寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	<p>SLV_RX_DATA_LOST: 从机接收数据丢失。该位指示当 IC_ENABLE 位 0 从 1 变为 0 时，从机接收操作因至少接收到一个 I2C 传输的数据字节而被中止。当读取为 1 时，DW_apb_i2c 被视为已主动参与被中止的 I2C 传输（匹配地址），且已进入 I2C 传输的数据阶段，即使数据字节已被响应为 NACK。</p> <p>注意：如果远程 I2C 主机在 DW_apb_i2c 有机会对传输发送 NACK 之前以 STOP 条件终止传输，且 IC_ENABLE[0] 已设置为 0，则该位亦被置为 1。</p> <p>当读取值为 0 时，DW_apb_i2c 被视为已禁用，且未主动介入从机接收传输的数据阶段。</p> <p>注意：仅当 IC_EN（第 0 位）读取为 0 时，CPU 才可安全读取此位。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：从机接收数据未丢失		
	0x1 → 活动：从机接收数据丢失		

Bits	Description	Type	Reset
1	<p>SLV_DISABLED_WHILE_BUSY: Slave Disabled While Busy (Transmit, Receive). This bit indicates if a potential or active Slave operation has been aborted due to the setting bit 0 of the IC_ENABLE register from 1 to 0. This bit is set when the CPU writes a 0 to the IC_ENABLE register while:</p> <p>(a) DW_apb_i2c is receiving the address byte of the Slave-Transmitter operation from a remote master;</p> <p>OR,</p> <p>(b) address and data bytes of the Slave-Receiver operation from a remote master.</p> <p>When read as 1, DW_apb_i2c is deemed to have forced a NACK during any part of an I2C transfer, irrespective of whether the I2C address matches the slave address set in DW_apb_i2c (IC_SAR register) OR if the transfer is completed before IC_ENABLE is set to 0 but has not taken effect.</p> <p>Note: If the remote I2C master terminates the transfer with a STOP condition before the DW_apb_i2c has a chance to NACK a transfer, and IC_ENABLE[0] has been set to 0, then this bit will also be set to 1.</p> <p>When read as 0, DW_apb_i2c is deemed to have been disabled when there is master activity, or when the I2C bus is idle.</p> <p>Note: The CPU can safely read this bit when IC_EN (bit 0) is read as 0.</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → INACTIVE: Slave is disabled when it is idle		
	0x1 → ACTIVE: Slave is disabled when it is active		
0	<p>IC_EN: ic_en Status. This bit always reflects the value driven on the output port ic_en. - When read as 1, DW_apb_i2c is deemed to be in an enabled state. - When read as 0, DW_apb_i2c is deemed completely inactive. Note: The CPU can safely read this bit anytime. When this bit is read as 0, the CPU can safely read SLV_RX_DATA_LOST (bit 2) and SLV_DISABLED_WHILE_BUSY (bit 1).</p> <p>Reset value: 0x0</p>	RO	0x0
	Enumerated values:		
	0x0 → DISABLED: I2C disabled		
	0x1 → ENABLED: I2C enabled		

I2C: IC_FS_SPKLEN Register

Offset: 0xa0

Description

I2C SS, FS or FM+ spike suppression limit

This register is used to store the duration, measured in ic_clk cycles, of the longest spike that is filtered out by the spike suppression logic when the component is operating in SS, FS or FM+ modes. The relevant I2C requirement is tSP (table

位	描述	类型	复位值
1	<p>SLV_DISABLED_WHILE_BUSY: 从机在忙碌期间被禁用（发送、接收）。该位指示是否因 IC_ENABLE 寄存器第 0 位由 1 变为 0 而导致潜在或正在进行的从机操作被中止。当 CPU 在以下情况下向 IC_ENABLE 寄存器写入 0 时，该位被置位：</p> <p>(a) DW_apb_i2c 正在接收来自远程主控的从设备发送操作的地址字节；</p> <p>或者，</p> <p>(b) 来自远程主控的从设备接收操作的地址字节和数据字节。</p> <p>当读取值为1时，无论I2C地址是否与DW_apb_i2c（IC_SAR寄存器）中设定的从设备地址匹配，或者传输是否在IC_ENABLE设置为0后但尚未生效之前完成，DW_apb_i2c均视为已在I2C传输的任何阶段强制产生NACK。</p> <p>注意：如果远程I2C主控在DW_apb_i2c有机会响应NACK之前以STOP条件终止传输，且IC_ENABLE[0]已被设置为0，则该位也将被设置为1。</p> <p>当读取值为0时，DW_apb_i2c视为在主控活动期间被禁用，或当I2C总线处于空闲状态。</p> <p>注意：仅当 IC_EN（第 0 位）读取为 0 时，CPU 才可安全读取此位。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 非活动：从设备在空闲时被禁用		
	0x1 → 活动：从设备在激活时被禁用		
0	<p>IC_EN: ic_en 状态。该位始终反映输出端口的驱动值 ic_en。- 读取为1时，DW_apb_i2c 被视为处于启用状态。- 读取为0时，DW_apb_i2c 被视为完全不活动。注意：CPU 可在任何时间安全读取该位。当该位读取为0时，CPU 可安全读取 SLV_RX_DATA_LOST（位2）和 SLV_DISABLE_WHILE_BUSY（位1）。</p> <p>复位值：0x0</p>	只读	0x0
	枚举值：		
	0x0 → 禁用：I2C 已禁用		
	0x1 → 启用：I2C 已启用		

I2C: IC_FS_SPKLEN 寄存器

偏移量：0xa0

描述

I2C SS、FS 或 FM+ 峰值抑制限制

该寄存器用于存储组件在 SS、FS 或 FM+ 模式下运行时，由脉冲抑制逻辑滤除的最长脉冲持续时间，单位为 ic_clk 时钟周期。相关的I2C要求是tSP（表

4) as detailed in the I2C Bus Specification. This register must be programmed with a minimum value of 1.

Table 490.
IC_FS_SPKLEN
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	IC_FS_SPKLEN: This register must be set before any I2C bus transaction can take place to ensure stable operation. This register sets the duration, measured in ic_clk cycles, of the longest spike in the SCL or SDA lines that will be filtered out by the spike suppression logic. This register can be written only when the I2C interface is disabled which corresponds to the IC_ENABLE[0] register being set to 0. Writes at other times have no effect. The minimum valid value is 1; hardware prevents values less than this being written, and if attempted results in 1 being set. For more information, refer to 'Spike Suppression'.	RW	0x07

I2C: IC_CLR_RESTART_DET Register

Offset: 0xa8

Description

Clear RESTART_DET Interrupt Register

Table 491.
IC_CLR_RESTART_DET
Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	CLR_RESTART_DET: Read this register to clear the RESTART_DET interrupt (bit 12) of IC_RAW_INTR_STAT register. Reset value: 0x0	RO	0x0

I2C: IC_COMP_PARAM_1 Register

Offset: 0xf4

Description

Component Parameter Register 1

Note This register is not implemented and therefore reads as 0. If it was implemented it would be a constant read-only register that contains encoded information about the component's parameter settings. Fields shown below are the settings for those parameters

Table 492.
IC_COMP_PARAM_1
Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:16	TX_BUFFER_DEPTH: TX Buffer Depth = 16	RO	0x00
15:8	RX_BUFFER_DEPTH: RX Buffer Depth = 16	RO	0x00
7	ADD_ENCODED_PARAMS: Encoded parameters not visible	RO	0x0
6	HAS_DMA: DMA handshaking signals are enabled	RO	0x0
5	INTR_IO: COMBINED Interrupt outputs	RO	0x0
4	HC_COUNT_VALUES: Programmable count values for each mode.	RO	0x0
3:2	MAX_SPEED_MODE: MAX SPEED MODE = FAST MODE	RO	0x0
1:0	APB_DATA_WIDTH: APB data bus width is 32 bits	RO	0x0

I2C: IC_COMP_VERSION Register

4) , 详见I2C总线规范。本寄存器必须设置为最小值1。

表490。
IC_FS_SPKLEN
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	IC_FS_SPKLEN: 必须在任何I2C总线事务发生前设置该寄存器, 以确保稳定运行。该寄存器以ic_clk周期为单位, 设定SCL或SDA线上最长尖峰的持续时间, 尖峰抑制逻辑将过滤该持续时间内的尖峰。仅当 I2C 接口被禁用时(对应 IC_ENABLE[0] 寄存器设为 0) , 才允许写入此寄存器。在其他时间写入无效。最小有效值为1; 硬件禁止写入小于该值的数值, 若尝试写入, 则自动设置为1。更多信息请参见“尖峰抑制”。	读写	0x07

I2C: IC_CLR_RESTART_DET寄存器

偏移: 0xa8

描述

清除 RESTART_DET 中断寄存器

表491。
IC_CLR_RESTART_DET
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	CLR_RESTART_DET: 读取该寄存器以清除IC_RAW_INTR_STAT寄存器中的RESTART_DET中断(第12位)。 复位值: 0x0	只读	0x0

I2C: IC_COMP_PARAM_1 寄存器

偏移: 0xf4

说明

组件参数寄存器 1

注意: 此寄存器未实现, 因此读取值恒为 0。若已实现, 该寄存器将为常量只读, 包含组件参数设置的编码信息。下表字段展示了相关参数的设定

表 492。
IC_COMP_PARAM_1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:16	TX_BUFFER_DEPTH: TX 缓冲区深度 = 16	只读	0x00
15:8	RX_BUFFER_DEPTH: RX 缓冲区深度 = 16	只读	0x00
7	ADD_ENCODED_PARAMS: 编码参数不可见	只读	0x0
6	HAS_DMA: DMA 握手信号已启用	只读	0x0
5	INTR_IO: 组合中断输出	只读	0x0
4	HC_COUNT_VALUES: 各模式的可编程计数值	只读	0x0
3:2	MAX_SPEED_MODE: 最大速度模式 = 快速模式	只读	0x0
1:0	APB_DATA_WIDTH: APB 数据总线宽度为 32 位	只读	0x0

I2C: IC_COMP_VERSION 寄存器

Offset: 0xf8**Description**

I2C Component Version Register

Table 493.
IC_COMP_VERSION
Register

Bits	Description	Type	Reset
31:0	IC_COMP_VERSION	RO	0x3230312a

I2C: IC_COMP_TYPE Register**Offset:** 0xfc**Description**

I2C Component Type Register

Table 494.
IC_COMP_TYPE
Register

Bits	Description	Type	Reset
31:0	IC_COMP_TYPE : Designware Component Type number = 0x44_57_01_40. This assigned unique hex value is constant and is derived from the two ASCII letters 'DW' followed by a 16-bit unsigned number.	RO	0x44570140

4.4. SPI

ARM Documentation

Excerpted from the [ARM PrimeCell Synchronous Serial Port \(PL022\) Technical Reference Manual](#). Used with permission.

RP2040 has two identical SPI controllers, both based on an ARM Primecell Synchronous Serial Port (SSP) (PL022) (Revision r1p4). Note this is NOT the same as the QSPI interface covered in [Section 4.10](#).

Each controller supports the following features:

- Master or Slave modes
 - Motorola SPI-compatible interface
 - Texas Instruments synchronous serial interface
 - National Semiconductor Microwire interface
- 8 deep Tx and Rx FIFOs
- Interrupt generation to service FIFOs or indicate error conditions
- Can be driven from DMA
- Programmable clock rate
- Programmable data size 4-16 bits

Each controller can be connected to a number of GPIO pins as defined in the GPIO muxing [Table 279](#) in [Section 2.19.2](#). Connections to the GPIO muxing are prefixed with the SPI instance name `spi0_` or `spi1_`, and include the following:

- clock `sclk` (connects to SSPCLKOUT in the following sections when the controller is operating in master mode, or SSPCLKIN when in slave mode)
- active low chip select or frame sync `ss_n` (referred to as SSPFSSOUT in the following sections)
- transmit data `tx` (referred to as SSPTXD in the following sections, noting that nSSPOE is NOT connected to the `tx` pad, so output data is not tristated by the SPI controller)

偏移: 0xf8

说明

I2C 组件版本寄存器

表 493。
IC_COMP_VERSION
寄存器

位	描述	类型	复位值
31:0	IC_COMP_VERSION	只读	0x3230312a

I2C: IC_COMP_TYPE 寄存器

偏移量: 0xfc

描述

I2C 组件类型寄存器

表 494。
IC_COMP_TYPE
寄存器

位	描述	类型	复位值
31:0	IC_COMP_TYPE : Designware 组件类型编号 = 0x44_57_01_40。该唯一十六进制值为常量，由两个 ASCII 字符 'DW' 及随后一个 16 位无符号数派生。	只读	0x44570140

4.4. SPI**ARM文档**

节录自 ARM PrimeCell 同步串行端口 (PL022) 技术参考手册，已获授权使用。

RP2040 配备两个相同的 SPI 控制器，均基于 ARM PrimeCell 同步串行端口 (SSP) (PL022) (修订版 r1p4)。注意，此接口与第 4.10 节中所述的 QSPI 接口不同。

每个控制器支持以下功能：

- 主模式或从模式
 - 兼容 Motorola SPI 接口
 - Texas Instruments 同步串行接口
 - National Semiconductor Microwire 接口
- 8 级深度的发送和接收 FIFO
- 用于服务 FIFO 或指示错误状态的中断生成
- 支持由 DMA 驱动
- 可编程时钟频率
- 可编程数据宽度，4 至 16 位

每个控制器可连接至多个 GPIO 引脚，详见第 2.19.2 节 GPIO 复用表 279。

GPIO 复用连接以 SPI 实例名称 `spi0_` 或 `spi1_` 为前缀，具体包括以下内容：

- 时钟 `sclk` (控制器作为主设备时连接至以下章节中的 SSPCLKOUT，作为从设备时连接至 SSPCLKIN)
- 低电平有效的片选或帧同步信号 `ss_n` (以下章节称为 SSPFSSOUT)
- 发送数据 `tx` (以下章节称为 SSPTXD，注意 nSSPOE 未连接至 `tx` 引脚，因此 SPI 控制器不会对输出数据进行三态控制)

- receive data `rd` (referred to as SSPRXD in the following sections)

The SPI TX pin function is wired to always assert the pad output enable, and is not driven from nSSPOE. When multiple SPI slaves are sharing a bus software would need to switch the output enable. This could be done by toggling `oeover` field of the relevant `iobank0.ctrl` register, or by switching GPIO function.

The SPI uses `clk_peri` as its reference clock for SPI timing, and is referred to as SSPCLK in the following sections. `clk_sys` is used as the bus clock, and is referred to as PCLK in the following sections (also see [Section 2.15.1](#)).

4.4.1. Overview

The PrimeCell SSP is a master or slave interface for synchronous serial communication with peripheral devices that have Motorola SPI, National Semiconductor Microwire, or Texas Instruments synchronous serial interfaces.

The PrimeCell SSP performs serial-to-parallel conversion on data received from a peripheral device. The CPU accesses data, control, and status information through the AMBA APB interface. The transmit and receive paths are buffered with internal FIFO memories enabling up to eight 16-bit values to be stored independently in both transmit and receive modes. Serial data is transmitted on SSPTXD and received on SSPRXD.

The PrimeCell SSP includes a programmable bit rate clock divider and prescaler to generate the serial output clock, SSPCLKOUT, from the input clock, SSPCLK. Bit rates are supported to 2MHz and higher, subject to choice of frequency for SSPCLK, and the maximum bit rate is determined by peripheral devices.

You can use the control registers SSPCR0 and SSPCR1 to program the PrimeCell SSP operating mode, frame format, and size.

The following individually maskable interrupts are generated:

- SSPTXINTR requests servicing of the transmit buffer
- SSPRXINTR requests servicing of the receive buffer
- SSPRORINTR indicates an overrun condition in the receive FIFO
- SSPRTINTR indicates that a timeout period expired while data was present in the receive FIFO.

A single combined interrupt is asserted if any of the individual interrupts are asserted and unmasked. This interrupt is connected to the processor interrupt controllers in RP2040.

In addition to the above interrupts, a set of DMA signals are provided for interfacing with a DMA controller.

Depending on the operating mode selected, the SSPFSSOUT output operates as:

- an active-HIGH frame synchronization output for Texas Instruments synchronous serial frame format
- an active-LOW slave select for SPI and Microwire.

4.4.2. Functional Description

- 接收数据 `rd` (以下章节称为SSPRXD)

SPI TX 引脚功能被设计为始终断言垫片输出使能信号，且不由 nSSPOE 进行驱动。当多个 SPI 从设备共享总线时，软件需切换输出使能信号。此操作可通过切换相关 `iobank0.ctrl` 寄存器的 oeover 字段，或通过切换 GPIO 功能实现。

SPI 使用 `clk_peri` 作为 SPI 定时的参考时钟，后续章节中称其为 SSPCLK。

`clk_sys` 用作总线时钟，后续章节中称其为 PCLK (另见第 2.15.1 节)。

4.4.1. 概述

PrimeCell SSP 是一主控或从属接口，用于与具备 Motorola SPI、National Semiconductor Microwire 或 Texas Instruments 同步串行接口的外围设备进行同步串行通信。

PrimeCell SSP 对从外围设备接收的数据执行串行向并行的转换。CPU 通过 AMBA APB 接口访问数据、控制及状态信息。发送和接收路径均采用内部 FIFO 存储器进行缓冲，使得在发送和接收模式下均可独立存储多达八个 16 位值。串行数据通过 SSPTXD 发送，并通过 SSPRXD 接收。

PrimeCell SSP 包含可编程的比特率时钟分频器和预分频器，根据输入时钟 SSPCLK 生成串行输出时钟 SSPCLKOUT。支持的比特率可达 2MHz 及以上，具体取决于 SSPCLK 的频率选择，最大比特率由外设设备决定。

您可以使用控制寄存器 SSPCR0 和 SSPCR1 来配置 PrimeCell SSP 的工作模式、帧格式及大小。

以下各个可屏蔽中断被触发：

- SSPTXINTR 请求服务发送缓冲区
- SSPRXINTR 请求服务接收缓冲区
- SSPRORINTR 表示接收 FIFO 出现溢出条件
- SSPRTINTR 表示在接收 FIFO 有数据存在时超时期限已到。

当任何单个中断被断言且未屏蔽时，会断言一个组合的单一中断。该中断连接至 RP2040 的处理器中断控制器。

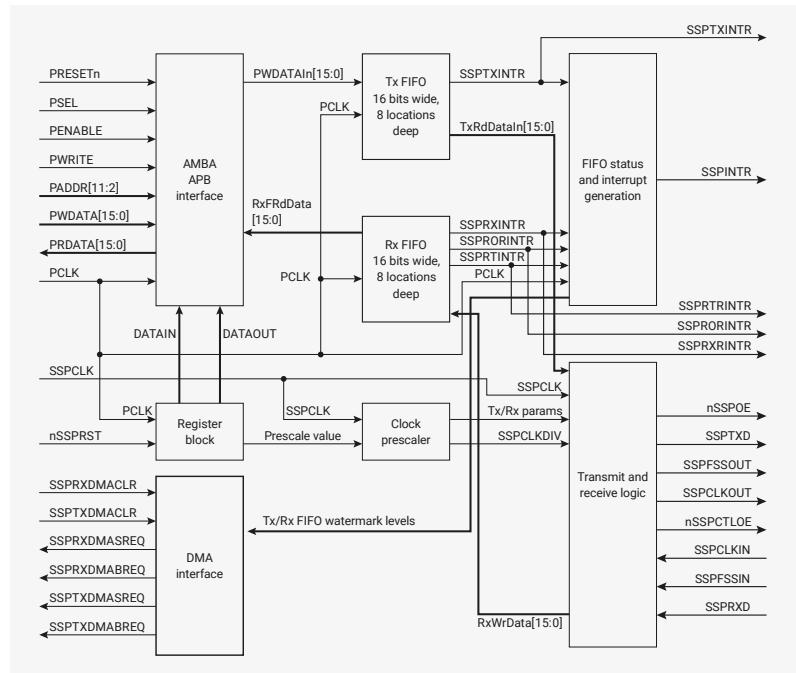
除了上述中断，还提供一组 DMA 信号，用于与 DMA 控制器接口。

根据所选工作模式，SSPFSSOUT 输出的功能为：

- 德州仪器同步串行帧格式的高电平有效帧同步输出。
- SPI 和 Microwire 的低电平有效从设备选择信号。

4.4.2. 功能描述

Figure 87. PrimeCell SSP block diagram.
For clarity, does not show the test logic.



4.4.2.1. AMBA APB interface

The AMBA APB interface generates read and write decodes for accesses to status and control registers, and transmit and receive FIFO memories.

4.4.2.2. Register block

The register block stores data written, or to be read, across the AMBA APB interface.

4.4.2.3. Clock prescaler

When configured as a master, an internal prescaler, comprising two free-running reloadable serially linked counters, provides the serial output clock SSPCLKOUT.

You can program the clock prescaler, using the SSPCPSR register, to divide SSPCLK by a factor of 2-254 in steps of two. By not utilizing the least significant bit of the SSPCPSR register, division by an odd number is not possible which ensures that a symmetrical, equal mark space ratio, clock is generated. See [SSPCPSR](#).

The output of the prescaler is divided again by a factor of 1-256, by programming the SSPCR0 control register, to give the final master output clock SSPCLKOUT.

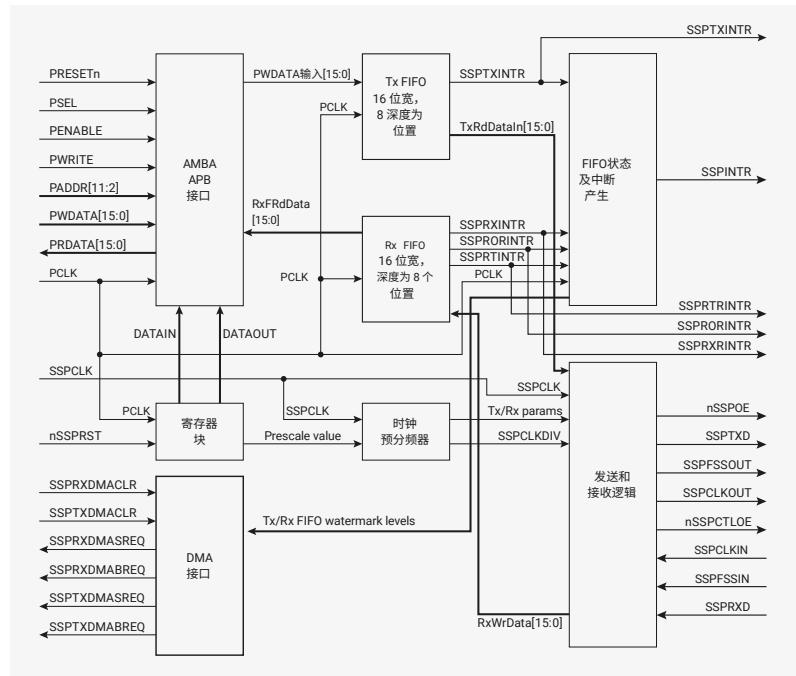
NOTE

The PCLK and SSPCLK clock inputs in [Figure 87](#) are connected to the `clk_sys` and `clk_peri` system-level clock nets on RP2040, respectively. By default `clk_peri` is attached directly to the system clock, but can be detached to maintain constant SPI frequency if the system clock is varied dynamically. See [Figure 28](#) for an overview of the RP2040 clock architecture.

4.4.2.4. Transmit FIFO

The common transmit FIFO is a 16-bit wide, 8-locations deep memory buffer. CPU data written across the AMBA APB

图87。PrimeCell
SSP模块示意图
为清晰起见，未
显示测试逻辑。



4.4.2.1 AMBA APB接口

AMBA APB接口生成用于访问状态和控制寄存器及传输与接收FIFO存储器的读写译码信号。

4.4.2.2 寄存器块

寄存器块存储通过AMBA APB接口写入或拟读取的数据。

4.4.2.3 时钟分频器

配置为主设备时，内部分频器由两个串联的可重载自由运行计数器组成，用以产生串行输出时钟SSPCLKOUT。

您可通过SSPCPSR寄存器设置时钟分频器，将SSPCLK按2至254之间的偶数因子分频。

因SSPCPSR寄存器未使用最低有效位，无法实现奇数分频，从而确保产生对称且占空比相等的时钟。详见SSPCPSR。

预分频器的输出通过编程SSPCRO控制寄存器，再次按1-256之间的因子分频，以产生最终的主控输出时钟SSPCLKOUT。

① 注意

图87中的PCLK和SSPCLK时钟输入分别连接至RP2040上的系统级时钟网`clk_sys`和`clk_peri`。默认情况下，`clk_peri`直接连接至系统时钟，但当系统时钟动态变化时，可解除连接以保持SPI频率恒定。有关RP2040时钟架构的概述，请参见图28。

4.4.2.4. 发送FIFO

通用发送FIFO为16位宽、深度为8个存储单元的缓冲器。通过AMBA APB总线写入的CPU数据

interface are stored in the buffer until read out by the transmit logic.

When configured as a master or a slave, parallel data is written into the transmit FIFO prior to serial conversion, and transmission to the attached slave or master respectively, through the SSPTXD pin.

4.4.2.5. Receive FIFO

The common receive FIFO is a 16-bit wide, 8-locations deep memory buffer. Received data from the serial interface are stored in the buffer until read out by the CPU across the AMBA APB interface.

When configured as a master or slave, serial data received through the SSPRXD pin is registered prior to parallel loading into the attached slave or master receive FIFO respectively.

4.4.2.6. Transmit and receive logic

When configured as a master, the clock to the attached slaves is derived from a divided-down version of SSPCLK through the previously described prescaler operations. The master transmit logic successively reads a value from its transmit FIFO and performs parallel to serial conversion on it. Then, the serial data stream and frame control signal, synchronized to SSPCLKOUT, are output through the SSPTXD pin to the attached slaves. The master receive logic performs serial to parallel conversion on the incoming synchronous SSPRXD data stream, extracting and storing values into its receive FIFO, for subsequent reading through the APB interface.

When configured as a slave, the SSPCLKIN clock is provided by an attached master and used to time its transmission and reception sequences. The slave transmit logic, under control of the master clock, successively reads a value from its transmit FIFO, performs parallel to serial conversion, then outputs the serial data stream and frame control signal through the slave SSPTXD pin. The slave receive logic performs serial to parallel conversion on the incoming SSPRXD data stream, extracting and storing values into its receive FIFO, for subsequent reading through the APB interface.

4.4.2.7. Interrupt generation logic

The PrimeCell SSP generates four individual maskable, active-HIGH interrupts. A combined interrupt output is generated as an OR function of the individual interrupt requests.

The transmit and receive dynamic data-flow interrupts, SSPTXINTR and SSPRXINTR, are separated from the status interrupts so that data can be read or written in response to the FIFO trigger levels.

4.4.2.8. DMA interface

The PrimeCell SSP provides an interface to connect to a DMA controller, see [Section 4.4.3.16](#).

4.4.2.9. Synchronizing registers and logic

The PrimeCell SSP supports both asynchronous and synchronous operation of the clocks, PCLK and SSPCLK. Synchronization registers and handshaking logic have been implemented, and are active at all times. Synchronization of control signals is performed on both directions of data flow, that is:

- from the PCLK to the SSPCLK domain
- from the SSPCLK to the PCLK domain.

接口被存储于缓冲区，直至被发送逻辑读取。

配置为主机或从机时，平行数据在串行转换前写入发送FIFO，并分别通过SSPTXD引脚发送至所连接的从机或主机。

4.4.2.5. 接收 FIFO

通用接收 FIFO 是一个 16 位宽、深度为 8 个单元的内存缓冲区。从串行接口接收的数据存储在缓冲区中，直至通过 AMBA APB 接口由 CPU 读取。

当配置为主设备或从设备时，经过 SSPRXD 引脚接收的串行数据会先寄存，然后分别并行加载到附属的从设备或主设备接收 FIFO 中。

4.4.2.6. 发送与接收逻辑

当配置为主设备时，供给附属从设备的时钟信号来自 SSPCLK 经过预分频器处理后的降频信号。主设备的发送逻辑依次从其发送 FIFO 读取数据，并对其执行并行转串行转换。随后，同步于 SSPCLKOUT 的串行数据流及帧控制信号通过 SSP TXD 引脚输出至附属从设备。主设备的接收逻辑对输入的同步 SSPRXD 串行数据流执行串行转并行转换，将数值提取并存储至其接收 FIFO，以便通过 APB 接口后续读取。

当配置为从设备时，SSPCLKIN 时钟由连接的主设备提供，并用于定时传输和接收序列。从设备发送逻辑在主时钟控制下，依次从其发送 FIFO 读取数据，执行并行转串行转换，然后通过从设备 SSPTXD 引脚输出串行数据流及帧控制信号。从设备接收逻辑对输入的 SSPRXD 数据流执行串行转并行转换，提取并存入接收 FIFO，供后续通过 APB 接口读取。

4.4.2.7. 中断生成逻辑

PrimeCell SSP 产生四个可屏蔽的高电平有效独立中断。组合中断输出由各独立中断请求经逻辑或（OR）运算生成。

发送与接收动态数据流中断 SSPTXINTR 和 SSPRXINTR 与状态中断分开，以便根据 FIFO 触发级别读写数据。

4.4.2.8. DMA 接口

PrimeCell SSP 提供连接 DMA 控制器的接口，详见第 4.4.3.16 节。

4.4.2.9. 寄存器与逻辑的同步

PrimeCell SSP 支持 PCLK 和 SSPCLK 时钟的异步及同步操作。

同步寄存器和握手机制已实现，并始终处于激活状态。控制信号的同步在数据流的双向均已执行，具体为：

- 从 PCLK 域到 SSPCLK 域
- 从 SSPCLK 域到 PCLK 域。

4.4.3. Operation

4.4.3.1. Interface reset

The PrimeCell SSP is reset by the global reset signal, PRESETn, and a block-specific reset signal, nSSPRST. The device reset controller asserts nSSPRST asynchronously and negate it synchronously to SSPCLK.

4.4.3.2. Configuring the SSP

Following reset, the PrimeCell SSP logic is disabled and must be configured when in this state. It is necessary to program control registers SSPCR0 and SSPCR1 to configure the peripheral as a master or slave operating under one of the following protocols:

- Motorola SPI
- Texas Instruments SSI
- National Semiconductor.

The bit rate, derived from the external SSPCLK, requires the programming of the clock prescale register SSPCPSR.

4.4.3.3. Enable PrimeCell SSP operation

You can either prime the transmit FIFO, by writing up to eight 16-bit values when the PrimeCell SSP is disabled, or permit the transmit FIFO service request to interrupt the CPU. Once enabled, transmission or reception of data begins on the transmit, SSPTXD, and receive, SSPRXD, pins.

4.4.3.4. Clock ratios

There is a constraint on the ratio of the frequencies of PCLK to SSPCLK. The frequency of SSPCLK must be less than or equal to that of PCLK. This ensures that control signals from the SSPCLK domain to the PCLK domain are guaranteed to get synchronized before one frame duration:

$$F_{SSPCLK} \leq F_{PCLK}$$

In the slave mode of operation, the SSPCLKIN signal from the external master is double-synchronized and then delayed to detect an edge. It takes three SSPCLKs to detect an edge on SSPCLKIN. SSPTXD has less setup time to the falling edge of SSPCLKIN on which the master is sampling the line.

The setup and hold times on SSPRXD, with reference to SSPCLKIN, must be more conservative to ensure that it is at the right value when the actual sampling occurs within the SSPMS. To ensure correct device operation, SSPCLK must be at least 12 times faster than the maximum expected frequency of SSPCLKIN.

The frequency selected for SSPCLK must accommodate the desired range of bit clock rates. The ratio of minimum SSPCLK frequency to SSPCLKOUT maximum frequency in the case of the slave mode is 12, and for the master mode, it is two.

For example, at the maximum SSPCLK (`clk_peri`) frequency on RP2040 of 133MHz, the maximum peak bit rate in master mode is 62.5Mbps. This is achieved with the SSPCPSR register programmed with a value of 2, and the SCR[7:0] field in the SSPCR0 register programmed with a value of 0.

In slave mode, the same maximum SSPCLK frequency of 133MHz can achieve a peak bit rate of $133 / 12 = \sim 11.083$ Mbps. The SSPCPSR register can be programmed with a value of 12, and the SCR[7:0] field in the SSPCR0 register can be programmed with a value of 0. Similarly, the ratio of SSPCLK maximum frequency to SSPCLKOUT minimum frequency is 254×256 .

The minimum frequency of SSPCLK is governed by the following inequalities, both of which must be satisfied:

4.4.3. 操作

4.4.3.1. 接口复位

PrimeCell SSP 通过全局复位信号 PRESETn 及模块专用复位信号 nSSPRST 进行复位。设备复位控制器异步置位 nSSPRST，并同步于 SSPCLK 时钟使其失效。

4.4.3.2. SSP 配置

复位后，PrimeCell SSP 逻辑被禁用，须在该状态下进行配置。必须编程控制寄存器 SSPCR0 和 SSPCR1，以配置外设为主机或从机，并使其在以下协议之一下工作：

- Motorola SPI
- Texas Instruments SSI
- National Semiconductor.

比特率由外部 SSPCLK 提供，需编程时钟预分频寄存器 SSPCPSR。

4.4.3.3. 启用 PrimeCell SSP 操作

您可以在 PrimeCell SSP 禁用时，通过写入最多八个16位值来预置发送 FIFO，或允许发送 FIFO 服务请求以中断 CPU。启用后，数据的传输或接收将通过传输引脚 SSPTXD 和接收引脚 SSPRXD 开始。

4.4.3.4. 时钟比率

PCLK 与 SSPCLK 频率比存在约束。SSPCLK 的频率必须小于或等于 PCLK 的频率，以确保从 SSPCLK 域到 PCLK 域的控制信号能在单帧周期内完成同步：

$$F_{SSPCLK} < F_{PCLK}$$

在从属模式下，来自外部主设备的 SSPCLKIN 信号经过双重同步后被延迟，以便检测边沿。检测 SSPCLKIN 上的边沿需三个 SSPCLK。SSPTXD 相对于主设备采样线路时的 SSPCLKIN 下降沿，其建立时间较短。

就 SSPCLKIN 而言，SSPRXD 的建立和保持时间必须更加保守，以确保在 SSPMS 内实际采样时值正确。为保证设备正常运行，SSPCLK 频率须至少为 SSPCLKIN 最大预期频率的 12 倍。

所选 SSPCLK 频率必须满足所需的位时钟速率范围。在从属模式下，SSPCLK 最低频率与 SSPCLKOUT 最大频率的比率为 12；在主模式下，该比率为 2。

例如，在 RP2040 上最大 SSPCLK (`clk_peri`) 频率为 133MHz 时，主模式的最大峰值比特率为 62.5Mbps。此速率通过将 SSPCPSR 寄存器设置为 2，且 SSPCR0 寄存器中的 SCR[7:0] 域设置为 0 实现。

在从属模式下，相同的最大 SSPCLK 频率 133MHz 可实现峰值比特率约为 $133 / 12 = 11.083$ Mbps。SSPCPSR 寄存器可设置为 12，SSPCR0 寄存器中的 SCR[7:0] 字段可设置为 0。同理，SSPCLK 最大频率与 SSPCLKOUT 最小频率的比值为 254×256 。

SSPCLK 的最低频率受以下不等式限制，且需同时满足：

$F_{SSPCLK}(min) \geq 2 \times F_{SSPCLKOUT}(max)$, for master mode

$F_{SSPCLK}(min) \geq 12 \times F_{SSPCLKIN}(max)$, for slave mode.

The maximum frequency of SSPCLK is governed by the following inequalities, both of which must be satisfied:

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKOUT}(min)$, for master mode

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKIN}(min)$, for slave mode.

4.4.3.5. Programming the SSPCR0 Control Register

The SSPCR0 register is used to:

- program the serial clock rate
- select one of the three protocols
- select the data word size, where applicable.

The Serial Clock Rate (SCR) value, in conjunction with the SSPCPSR clock prescale divisor value, CPSDVSR, is used to derive the PrimeCell SSP transmit and receive bit rate from the external SSPCLK.

The frame format is programmed through the FRF bits, and the data word size through the DSS bits.

Bit phase and polarity, applicable to Motorola SPI format only, are programmed through the SPH and SPO bits.

4.4.3.6. Programming the SSPCR1 Control Register

The SSPCR1 register is used to:

- select master or slave mode
- enable a loop back test feature
- enable the PrimeCell SSP peripheral.

To configure the PrimeCell SSP as a master, clear the SSPCR1 register master or slave selection bit, MS, to 0. This is the default value on reset.

Setting the SSPCR1 register MS bit to 1 configures the PrimeCell SSP as a slave. When configured as a slave, enabling or disabling of the PrimeCell SSP SSPTXD signal is provided through the SSPCR1 slave mode SSPTXD output disable bit, SOD. You can use this in some multi-slave environments where masters might parallel broadcast.

To enable the operation of the PrimeCell SSP, set the Synchronous Serial Port Enable (SSE) bit to 1.

4.4.3.6.1. Bit rate generation

The serial bit rate is derived by dividing down the input clock, SSPCLK. The clock is first divided by an even prescale value CPSDVSR in the range 2-254, and is programmed in SSPCPSR. The clock is divided again by a value in the range 1-256, that is 1 + SCR, where SCR is the value programmed in SSPCR0.

The following equation defines the frequency of the output signal bit clock, SSPCLKOUT:

$$F_{SSPCLKOUT} = \frac{F_{SSPCLK}}{CPSDVSR \times (1 + SCR)}$$

For example, if SSPCLK is 125MHz, and CPSDVSR = 2, then SSPCLKOUT has a frequency range from 244kHz - 62.5MHz.

$F_{SSPCLK}(min) \geq 2 \times F_{SSPCLKOUT}(max)$, 适用于主模式

$F_{SSPCLK}(min) \geq 12 \times F_{SSPCLKIN}(max)$, 适用于从模式。

SSPCLK 的最高频率受以下不等式限制，且需同时满足：

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKOUT}(min)$, 适用于主模式

$F_{SSPCLK}(max) \leq 254 \times 256 \times F_{SSPCLKIN}(min)$, 适用于从模式。

4.4.3.5. SSPCR0 控制寄存器的编程

SSPCR0 寄存器用于：

- 设置串行时钟速率
- 选择三种协议之一
- 选择数据字大小（如适用）

串行时钟速率（SCR）值结合 SSPCPSR 时钟预分频除数 CPSDVS，用于从外部 SSPCLK 推导 PrimeCell SSP 的发送和接收比特率。

帧格式通过FRF位进行编程，数据字大小通过DSS位进行编程。

位相和极性，仅适用于Motorola SPI格式，通过SPH和SPO位进行编程。

4.4.3.6. SSPCR1控制寄存器的编程

SSPCR1寄存器用于：

- 选择主模式或从模式
- 启用环回测试功能
- 启用PrimeCell SSP外设。

要将PrimeCell SSP配置为主模式，请将SSPCR1寄存器中的主从选择位MS清零。该值为复位时的默认值。

将SSPCR1寄存器的MS位置1可将PrimeCell SSP配置为从模式。配置为从模式时，PrimeCell SSP的SSPTXD信号的启用或禁用由SSPCR1寄存器中从模式SSPTXD输出禁用位SOD控制。此功能在某些多从设备环境中有用，其中主设备可能进行并行广播。

要启用 PrimeCell SSP 的功能，请将同步串行端口使能位（SSE）设置为 1。

4.4.3.6.1. 比特率生成

串行比特率通过对输入时钟 SSPCLK 进行分频获得。时钟首先由一个在 2 至 254 范围内的偶数预分频值 CPSDVS 除，该值在 SSPCPSR 寄存器中配置。随后时钟再次被一个 1 至 256 范围内的值除，即 $1 + SCR$ ，其中 SCR 是在 SSPCR0 中编程设置的值。

以下公式定义了输出信号比特时钟 SSPCLKOUT 的频率：

$$F_{SSPCLKOUT} = \frac{F_{SSPCLK}}{CPSDVS \times (1 + SCR)}$$

例如，若 SSPCLK 为 125MHz，且 CPSDVS=2，则 SSPCLKOUT 的频率范围为 244kHz 至 62.5MHz。

4.4.3.7. Frame format

Each data frame is between 4-16 bits long, depending on the size of data programmed, and is transmitted starting with the MSB. You can select the following basic frame types:

- Texas Instruments synchronous serial
- Motorola SPI
- National Semiconductor Microwire.

For all formats, the serial clock, SSPCLKOUT, is held inactive while the PrimeCell SSP is idle, and transitions at the programmed frequency only during active transmission or reception of data. The idle state of SSPCLKOUT is utilized to provide a receive timeout indication that occurs when the receive FIFO still contains data after a timeout period.

For Motorola SPI and National Semiconductor Microwire frame formats, the serial frame, SSPFSSOUT, pin is active-LOW, and is asserted, pulled-down, during the entire transmission of the frame.

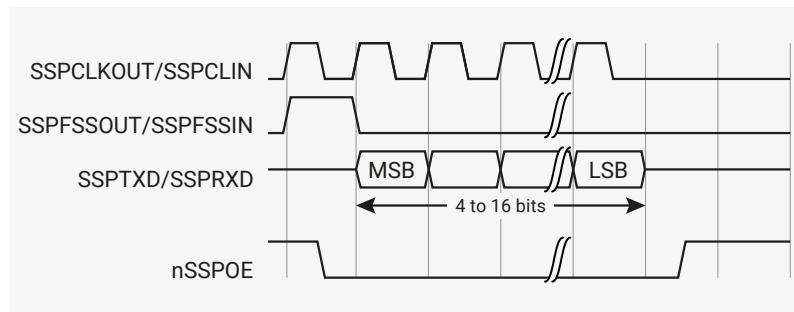
For Texas Instruments synchronous serial frame format, the SSPFSSOUT pin is pulsed for one serial clock period, starting at its rising edge, prior to the transmission of each frame. For this frame format, both the PrimeCell SSP and the off-chip slave device drive their output data on the rising edge of SSPCLKOUT, and latch data from the other device on the falling edge.

Unlike the full-duplex transmission of the other two frame formats, the National Semiconductor Microwire format uses a special master-slave messaging technique that operates at half-duplex. In this mode, when a frame begins, an 8-bit control message is transmitted to the off-chip slave. During this transmit, the SSS receives no incoming data. After the message has been sent, the off-chip slave decodes it and, after waiting one serial clock after the last bit of the 8-bit control message has been sent, responds with the requested data. The returned data can be 4-16 bits in length, making the total frame length in the range 13-25 bits.

4.4.3.8. Texas Instruments synchronous serial frame format

[Figure 88](#) shows the Texas Instruments synchronous serial frame format for a single transmitted frame.

Figure 88. Texas Instruments synchronous serial frame format, single transfer



In this mode, SSPCLKOUT and SSPFSSOUT are forced LOW, and the transmit data line, SSPTXD is tristated whenever the PrimeCell SSP is idle. When the bottom entry of the transmit FIFO contains data, SSPFSSOUT is pulsed HIGH for one SSPCLKOUT period. The value to be transmitted is also transferred from the transmit FIFO to the serial shift register of the transmit logic. On the next rising edge of SSPCLKOUT, the MSB of the 4-bit to 16-bit data frame is shifted out on the SSPTXD pin. In a similar way, the MSB of the received data is shifted onto the SSPRXD pin by the off-chip serial slave device.

Both the PrimeCell SSP and the off-chip serial slave device then clock each data bit into their serial shifter on the falling edge of each SSPCLKOUT. The received data is transferred from the serial shifter to the receive FIFO on the first rising edge of PCLK after the LSB has been latched.

[Figure 89](#) shows the Texas Instruments synchronous serial frame format when back-to-back frames are transmitted.

4.4.3.7. 帧格式

每个数据帧长度介于 4 至 16 位之间，具体取决于配置的数据大小，传输时从最高有效位（MSB）开始。您可以选择以下基本帧类型：

- 德州仪器同步串行
- Motorola SPI
- 国家半导体Microwire。

对于所有格式，当PrimeCell SSP处于空闲状态时，串行时钟SSPCLKOUT保持非激活状态，仅在数据传输或接收期间以预定频率转换。SSPCLKOUT的空闲状态用于提供接收超时指示，当接收FIFO在超时期限后仍包含数据时触发该指示。

对于摩托罗拉SPI和国家半导体Microwire帧格式，串行帧信号SSPFSSOUT引脚为低有效，且在整个帧传输期间被拉低。

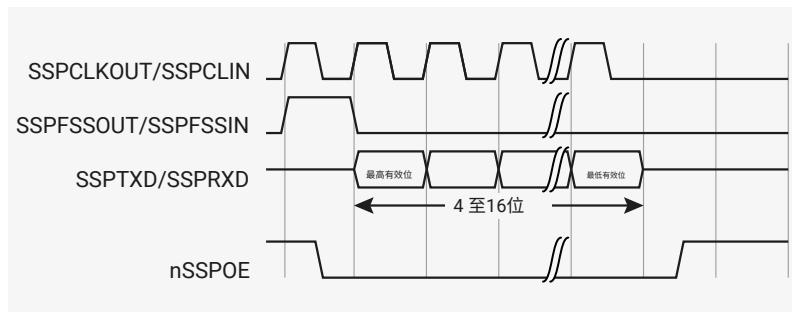
对于德州仪器同步串行帧格式，SSPFSSOUT引脚在每个帧传输开始前的上升沿触发一个串行时钟周期的脉冲。对于该帧格式，PrimeCell SSP及片外从设备均在SSPCLKOUT的上升沿驱动其输出数据，并在下降沿锁存对方设备的数据。

与另外两种帧格式的全双工传输不同，National Semiconductor Microwire 格式采用一种特殊的主从消息传递技术，工作于半双工模式。在该模式下，帧开始时会向芯片外的从设备发送一条8位控制消息。在此传输过程中，SSS 不接收任何输入数据。消息发送完成后，芯片外的从设备对其进行解码，并在最后一位8位控制消息发送结束后等待一个串行时钟周期，然后响应所请求数据。返回的数据长度可为4至16位，故总帧长度介于13至25位之间。

4.4.3.8. Texas Instruments同步串行帧格式

[图88展示了Texas Instruments单帧传输的同步串行帧格式。](#)

图88。德州仪器同步串行帧格式，单次传输

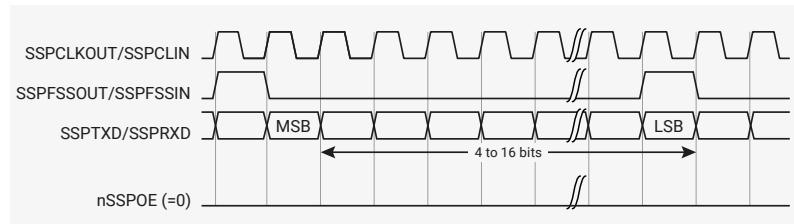


在此模式下，SSPCLKOUT 和 SSPFSSOUT 被强制拉低，当 PrimeCell SSP 空闲时，发送数据线 SSPTXD 处于三态。当发送 FIFO 底部条目包含数据时，SSPFSSOUT 会在一个 SSPCLKOUT 周期内被脉冲拉高。要传输的数值同时从发送 FIFO 转移至传输逻辑的串行移位寄存器。在下一次 SSPCLKOUT 上升沿时，4位至16位数据帧的最高有效位通过 SSPTXD 引脚移出。类似地，芯片外串行从设备将接收数据的最高有效位移入 SSPRXD 引脚。

PrimeCell SSP 与芯片外串行从设备均在每个 SSPCLKOUT 下降沿对各自串行移位寄存器中的数据位进行时钟驱动。接收数据在最低有效位锁存后，于 PCLK 首个上升沿从串行移位寄存器转移至接收 FIFO。

[图89展示了德州仪器连续传输的同步串行帧格式。](#)

Figure 89. Texas Instruments synchronous serial frame format, continuous transfer



4.4.3.9. Motorola SPI frame format

The Motorola SPI interface is a four-wire interface where the SSPFSSOUT signal behaves as a slave select. The main feature of the Motorola SPI format is that you can program the inactive state and phase of the SSPCLKOUT signal using the SPO and SPH bits of the SSPSCRO control register.

4.4.3.9.1. SPO, clock polarity

When the SPO clock polarity control bit is LOW, it produces a steady state LOW value on the SSPCLKOUT pin. If the SPO clock polarity control bit is HIGH, a steady state HIGH value is placed on the SSPCLKOUT pin when data is not being transferred.

4.4.3.9.2. SPH, clock phase

The SPH control bit selects the clock edge that captures data and enables it to change state. It has the most impact on the first bit transmitted by either permitting or not permitting a clock transition before the first data capture edge.

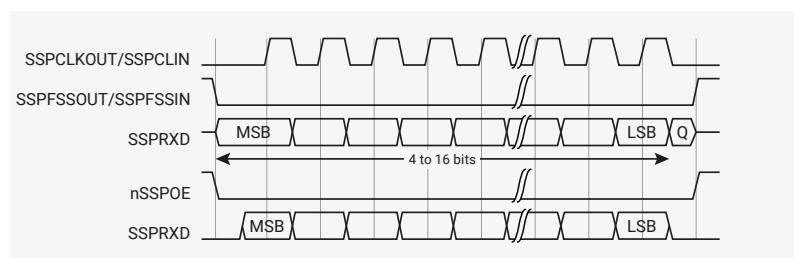
When the SPH phase control bit is LOW, data is captured on the first clock edge transition.

When the SPH clock phase control bit is HIGH, data is captured on the second clock edge transition.

4.4.3.10. Motorola SPI Format with SPO=0, SPH=0

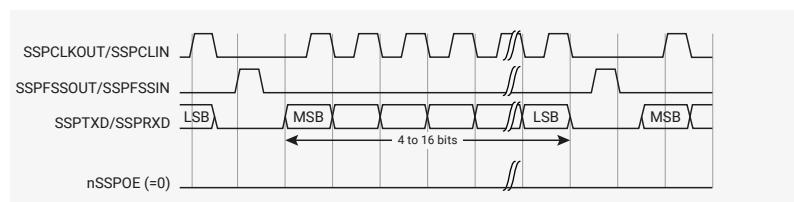
[Figure 90](#) and [Figure 91](#) shows a continuous transmission signal sequence for Motorola SPI frame format with SPO=0, SPH=0. [Figure 90](#) shows a single transmission signal sequence for Motorola SPI frame format with SPO=0, SPH=0.

Figure 90. Motorola SPI frame format, single transfer, with SPO=0 and SPH=0



[Figure 91](#) shows a continuous transmission signal sequence for Motorola SPI frame format with SPO=0, SPH=0.

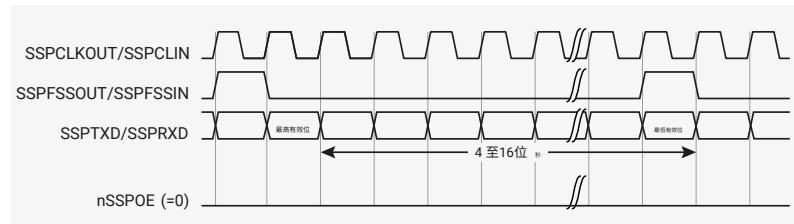
Figure 91. Motorola SPI frame format, single transfer, with SPO=0 and SPH=0



In this configuration, during idle periods:

- the SSPCLKOUT signal is forced LOW

图89。德州仪器同步串行帧格式，连续传输



4.4.3.9. Motorola SPI帧格式

Motorola SPI接口为四线接口，其中SSPFSSOUT信号用作从设备选择信号。Motorola SPI格式的主要特征是可通过SSPCRO控制寄存器中的SPO和SPH位来编程设置SSPCLKOUT信号的非活动状态和相位。

4.4.3.9.1. SPO，时钟极性

当SPO时钟极性控制位为低时，SSPCLKOUT引脚保持稳定的低电平；当SPO时钟极性控制位为高时，数据未传输期间，SSPCLKOUT引脚保持稳定的高电平。

4.4.3.9.2. SPH，时钟相位

SPH控制位用于选择捕获数据的时钟边沿，并允许其状态改变。该控制位对第一个传输的数据位影响最大，决定是否允许在第一个数据捕获边沿之前发生时钟跳变。

当SPH相位控制位为低电平时，数据在第一个时钟跳变边沿被捕获。

当SPH时钟相位控制位为高电平时，数据在第二个时钟跳变边沿被捕获。

4.4.3.10. Motorola SPI 格式，SPO=0，SPH=0

图90与图91展示了SPO=0，SPH=0条件下Motorola SPI帧格式的连续传输信号序列。图90展示了SPO=0，SPH=0条件下的单次传输信号序列。

图90。Motorola SPI帧格式，单次传输， $SPO=0$, $SPH=0$

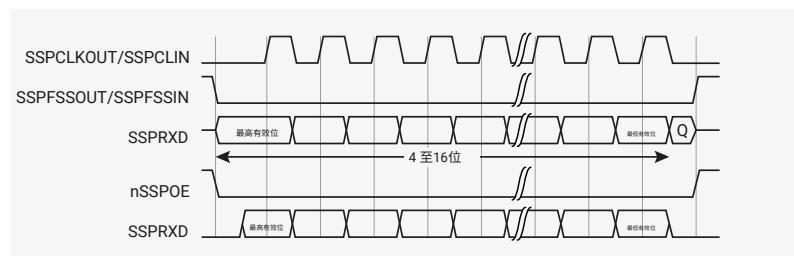
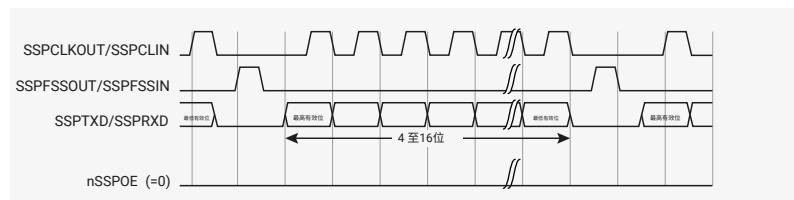


图91展示了SPO=0且SPH=0的Motorola SPI帧格式下的连续传输信号序列。

图91。Motorola SPI帧格式，单次传输， $SPO=0$ 且 $SPH=0$



在此配置下，空闲期间：

- SSPCLKOUT信号被强制保持低电平

- the SSPFSSOUT signal is forced HIGH
- the transmit data line SSPTXD is arbitrarily forced LOW
- the nSSPOE pad enable signal is forced HIGH (note this is not connected to the pad in RP2040)
- when the PrimeCell SSP is configured as a master, the nSSPCTLOE line is driven LOW, enabling the SSPCLKOUT pad, active-LOW enable
- when the PrimeCell SSP is configured as a slave, the nSSPCTLOE line is driven HIGH, disabling the SSPCLKOUT pad, active-LOW enable.

If the PrimeCell SSP is enable, and there is valid data within the transmit FIFO, the start of transmission is signified by the SSPFSSOUT master signal being driven LOW. This causes slave data to be enabled onto the SSPRXD input line of the master. The nSSPOE line is driven LOW, enabling the master SSPTXD output pad.

One-half SSPCLKOUT period later, valid master data is transferred to the SSPTXD pin. Now that both the master and slave data have been set, the SSPCLKOUT master clock pin goes HIGH after one additional half SSPCLKOUT period.

The data is now captured on the rising and propagated on the falling edges of the SSPCLKOUT signal.

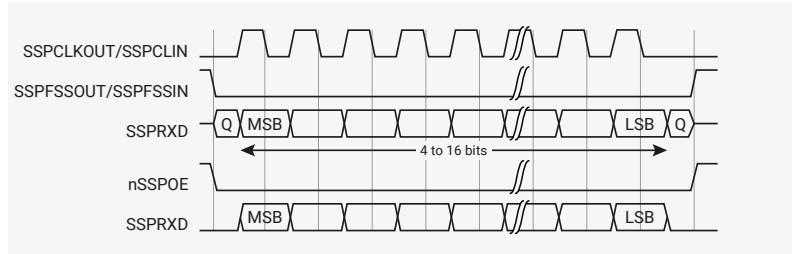
In the case of a single word transmission, after all bits of the data word have been transferred, the SSPFSSOUT line is returned to its idle HIGH state one SSPCLKOUT period after the last bit has been captured.

However, in the case of continuous back-to-back transmissions, the SSPFSSOUT signal must be pulsed HIGH between each data word transfer. This is because the slave select pin freezes the data in its serial peripheral register and does not permit it to be altered if the SPH bit is logic zero. Therefore, the master device must raise the SSPFSSIN pin of the slave device between each data transfer to enable the serial peripheral data write. On completion of the continuous transfer, the SSPFSSOUT pin is returned to its idle state one SSPCLKOUT period after the last bit has been captured.

4.4.3.11. Motorola SPI Format with SPO=0, SPH=1

Figure 92 shows the transfer signal sequence for Motorola SPI format with SPO=0, SPH=1, and it covers both single and continuous transfers.

Figure 92. Motorola SPI frame format with SPO=0 and SPH=1, single and continuous transfers



In this configuration, during idle periods:

- the SSPCLKOUT signal is forced LOW
- The SSPFSSOUT signal is forced HIGH
- the transmit data line SSPTXD is arbitrarily forced LOW
- the nSSPOE pad enable signal is forced HIGH (note this is not connected to the pad in RP2040)
- when the PrimeCell SSP is configured as a master, the nSSPCTLOE line is driven LOW, enabling the SSPCLKOUT pad, active-LOW enable
- when the PrimeCell SSP is configured as a slave, the nSSPCTLOE line is driven HIGH, disabling the SSPCLKOUT pad, active-LOW enable.

If the PrimeCell SSP is enabled, and there is valid data within the transmit FIFO, the start of transmission is signified by the SSPFSSOUT master signal being driven LOW. The nSSPOE line is driven LOW, enabling the master SSPTXD output pad. After an additional one half SSPCLKOUT period, both master and slave valid data is enabled onto their respective transmission lines. At the same time, the SSPCLKOUT is enabled with a rising edge transition.

- SSPFSSOUT信号被强制保持高电平
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

若启用PrimeCell SSP且传输FIFO内含有效数据，传输开始由主设备信号SSPFSSOUT拉低表示，此时从设备数据被使能至主设备的SSPRXD输入线。nSSPOE 线路被驱动为低电平，从而使主设备 SSPTXD 输出引脚生效。

半个 SSPCLKOUT 周期后，有效的主设备数据被传输至 SSPTXD 引脚。既然主设备和从设备的数据均已设定，SSPCLKOUT 主时钟引脚将在额外半个 SSPCLKOUT 周期后变为高电平。

数据现于 SSPCLKOUT 信号的上升沿被捕获，并于下降沿被传播。

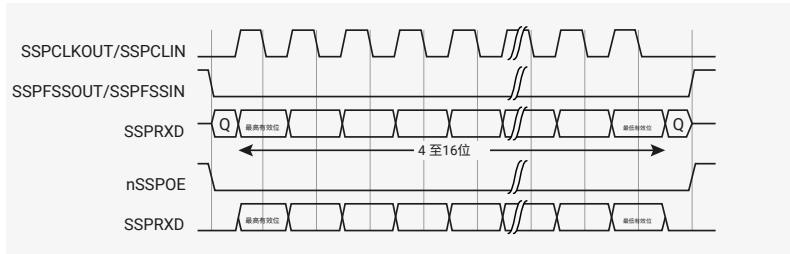
在单字传输情况下，所有数据字比特传输完成后，SSPFSSOUT 线将在最后一个比特捕获后一个 SSPCLKOUT 周期内恢复至其空闲高电平状态。

然而，在连续背靠背的传输中，必须在每个数据字传输之间对 SSPFSSOUT 信号施加高电平脉冲。这是因为从设备选择引脚会冻结串行外设寄存器中的数据，若 SPH 位为逻辑零，则不允许数据被更改。因此，主设备必须在每次数据传输之间拉高从设备的 SSPFSIN 引脚，以启用串行外设数据写入。连续传输结束后，最后一位被捕获后一个 SSPCLKOUT 周期，SSPFSSOUT 引脚返回其空闲状态。

4.4.3.11. SPO=0, SPH=1 的 Motorola SPI 格式

图 92 显示了 SPO=0, SPH=1 的 Motorola SPI 格式传输信号序列，涵盖单次及连续传输。

图 92。SPO=0 且
SPH=1 的 Motorola S
PI 帧格式，单次
及连续传输



在此配置下，空闲期间：

- SSPCLKOUT信号被强制保持低电平
- SSPFSSOUT 信号被强制拉高。
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

如果 PrimeCell SSP 启用且传输 FIFO 中存在有效数据，传输起始由 SSPFSSOUT 主控信号拉低表示。nSSPOE 线被拉低，启用主控 SSPTXD 输出引脚。在额外一个半 SSPCLKOUT 周期后，主从设备的有效数据同步使能于各自传输线。同时，SSPCLKOUT 在上升沿触发时被使能。

Data is then captured on the falling edges and propagated on the rising edges of the SSPCLKOUT signal.

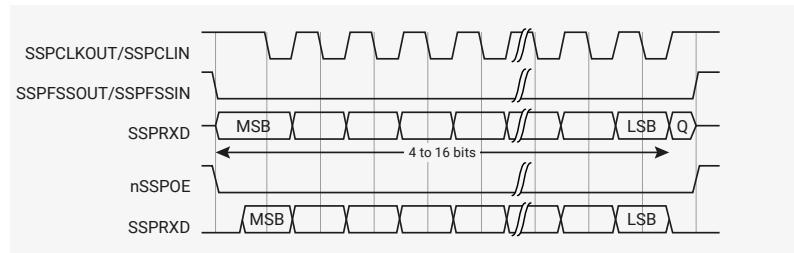
In the case of a single word transfer, after all bits have been transferred, the SSPFSSOUT line is returned to its idle HIGH state one SSPCLKOUT period after the last bit has been captured. For continuous back-to-back transfers, the SSPFSSOUT pin is held LOW between successive data words and termination is the same as that of the single word transfer.

4.4.3.12. Motorola SPI Format with SPO=1, SPH=0

[Figure 93](#) and [Figure 94](#) show single and continuous transmission signal sequences for Motorola SPI format with SPO=1, SPH=0.

[Figure 93](#) shows a single transmission signal sequence for Motorola SPI format with SPO=1, SPH=0.

Figure 93. Motorola SPI frame format, single transfer, with SPO=1 and SPH=0

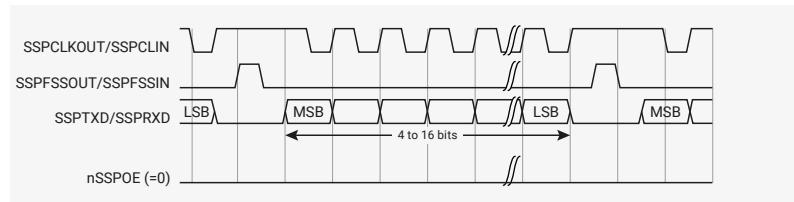


[Figure 94](#) shows a continuous transmission signal sequence for Motorola SPI format with SPO=1, SPH=0.

NOTE

In [Figure 93](#), Q is an undefined signal.

Figure 94. Motorola SPI frame format, continuous transfer, with SPO=1 and SPH=0



In this configuration, during idle periods:

- the SSPCLKOUT signal is forced HIGH
- the SSPFSSOUT signal is forced HIGH
- the transmit data line SSPTXD is arbitrarily forced LOW
- the nSSPOE pad enable signal is forced HIGH (note this is not connected to the pad in RP2040)
- when the PrimeCell SSP is configured as a master, the nSSPCTLOE line is driven LOW, enabling the SSPCLKOUT pad, active-LOW enable
- when the PrimeCell SSP is configured as a slave, the nSSPCTLOE line is driven HIGH, disabling the SSPCLKOUT pad, active-LOW enable.

If the PrimeCell SSP is enabled, and there is valid data within the transmit FIFO, the start of transmission is signified by the SSPFSSOUT master signal being driven LOW, and this causes slave data to be immediately transferred onto the SSPRXD line of the master. The nSSPOE line is driven LOW, enabling the master SSPTXD output pad.

One half period later, valid master data is transferred to the SSPTXD line. Now that both the master and slave data have been set, the SSPCLKOUT master clock pin becomes LOW after one additional half SSPCLKOUT period. This means that data is captured on the falling edges and be propagated on the rising edges of the SSPCLKOUT signal.

In the case of a single word transmission, after all bits of the data word are transferred, the SSPFSSOUT line is returned to its idle HIGH state one SSPCLKOUT period after the last bit has been captured.

数据随后在 SSPCLKOUT 信号的下降沿被采集，并在上升沿传输。

对于单字传输，在所有位传输完成后，SSPFSSOUT 线将在最后一位被采集后的一个 SSPCLKOUT 周期内回到其空闲的高电平 (HIGH) 状态。对于连续的背靠背传输，SSPFSSOUT 引脚在连续数据字之间保持低电平 (LOW)，其终止方式与单字传输相同。

4.4.3.12. Motorola SPI 格式，SPO=1，SPH=0

图93和图94展示了 SPO=1、SPH=0 的 Motorola SPI 格式下的单次及连续传输信号序列。

图93展示了 SPO=1、SPH=0 的 Motorola SPI 格式下的单次传输信号序列。

图93。Motorola S PI帧格式，单次传输，SPO=1且SPH=0

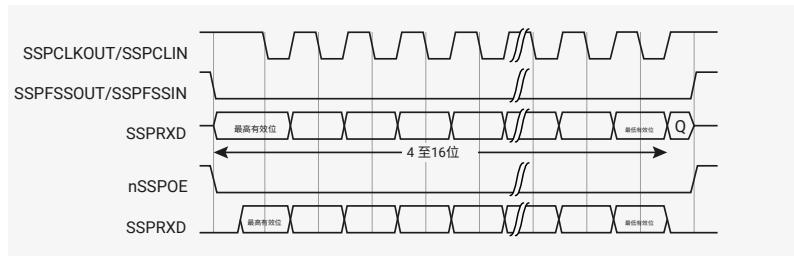
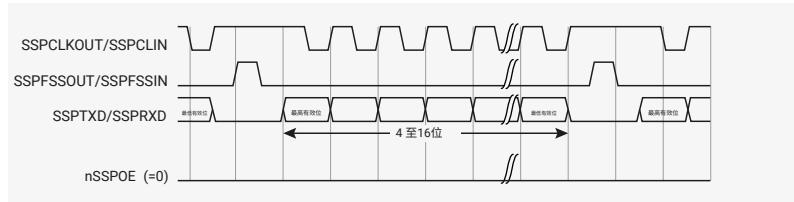


图94展示了SPO=1、SPH=0的Motorola SPI格式下的连续传输信号序列。

i 注意

图93中信号Q未定义。

图94。Motorola S PI帧格式，连续传输，SPO=1且SPH=0。



在此配置下，空闲期间：

- SSPCLKOUT信号被强制置为高电平。
- SSPFSSOUT信号被强制保持高电平
- 传输数据线SSPTXD被任意强制保持低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）
- 当PrimeCell SSP配置为主设备时，nSSPCTLOE线被拉低，启用SSPCLKOUT引脚，该使能信号为低有效
- 当PrimeCell SSP配置为从设备时，nSSPCTLOE线被拉高，禁用SSPCLKOUT引脚，该使能信号为低有效。

若启用PrimeCell SSP且发送FIFO中存在有效数据，则传输起始由SSPFSSOUT主控信号拉低表示，导致从设备数据立即传输至主设备的SSPRXD线上。nSSPOE 线路被驱动为低电平，从而使主设备 SSPTXD 输出引脚生效。

半个周期后，有效的主设备数据传输至SSPTXD线上。主从设备数据均设置完毕后，SSPCLKOUT主时钟引脚在额外半个SSPCLKOUT周期后变为低电平。这表示数据在SSPCLKOUT信号的下降沿被捕获，并在上升沿传播。

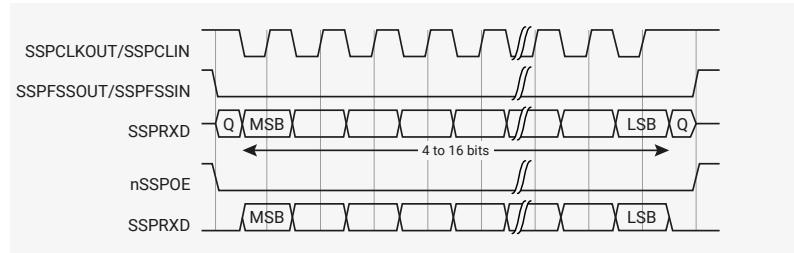
单字传输时，当数据字所有位传输完成后，SSPFSSOUT线于最后一位捕获后一个SSPCLKOUT周期内恢复至其空闲高电平状态。

However, in the case of continuous back-to-back transmissions, the SSPFSSOUT signal must be pulsed HIGH between each data word transfer. This is because the slave select pin freezes the data in its serial peripheral register and does not permit it to be altered if the SPH bit is logic zero. Therefore, the master device must raise the SSPFSSIN pin of the slave device between each data transfer to enable the serial peripheral data write. On completion of the continuous transfer, the SSPFSSOUT pin is returned to its idle state one SSPCLKOUT period after the last bit has been captured.

4.4.3.13. Motorola SPI Format with SPO=1, SPH=1

[Figure 95](#) shows the transfer signal sequence for Motorola SPI format with SPO=1, SPH=1, and it covers both single and continuous transfers.

Figure 95. Motorola SPI frame format with SPO=1 and SPH=1, single and continuous transfers



NOTE

In [Figure 95](#), Q is an undefined signal.

In this configuration, during idle periods:

- the SSPCLKOUT signal is forced HIGH
- the SSPFSSOUT signal is forced HIGH
- the transmit data line SSPTXD is arbitrarily forced LOW
- the nSSPOE pad enable signal is forced HIGH (note this is not connected to the pad in RP2040)
- when the PrimeCell SSP is configured as a master, the nSSPCTLOE line is driven LOW, enabling the SSPCLKOUT pad, active-LOW enable
- when the PrimeCell SSP is configured as a slave, the nSSPCTLOE line is driven HIGH, disabling the SSPCLKOUT pad, active-LOW enable.

If the PrimeCell SSP is enabled, and there is valid data within the transmit FIFO, the start of transmission is signified by the SSPFSSOUT master signal being driven LOW. The nSSPOE line is driven LOW, enabling the master SSPTXD output pad. After an additional one half SSPCLKOUT period, both master and slave data are enabled onto their respective transmission lines. At the same time, the SSPCLKOUT is enabled with a falling edge transition. Data is then captured on the rising edges and propagated on the falling edges of the SSPCLKOUT signal.

After all bits have been transferred, in the case of a single word transmission, the SSPFSSOUT line is returned to its idle HIGH state one SSPCLKOUT period after the last bit has been captured.

For continuous back-to-back transmissions, the SSPFSSOUT pin remains in its active-LOW state, until the final bit of the last word has been captured, and then returns to its idle state as the previous section describes.

For continuous back-to-back transfers, the SSPFSSOUT pin is held LOW between successive data words and termination is the same as that of the single word transfer.

4.4.3.14. National Semiconductor Microwire frame format

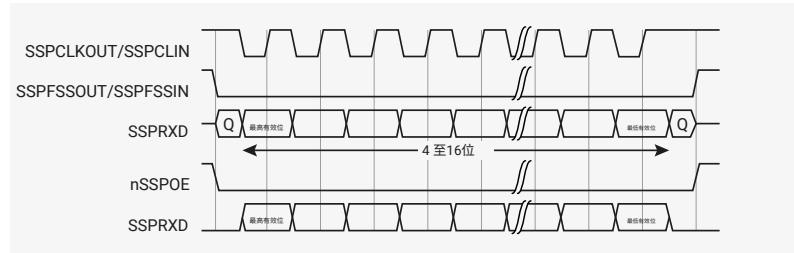
[Figure 96](#) shows the National Semiconductor Microwire frame format for a single frame. [Figure 97](#) shows the same format when back to back frames are transmitted.

然而，在连续背靠背的传输中，必须在每个数据字传输之间对 SSPFSSOUT 信号施加高电平脉冲。这是因为从设备选择引脚会冻结串行外设寄存器中的数据，若 SPH 位为逻辑零，则不允许数据被更改。因此，主设备必须在每次数据传输之间拉高从设备的 SSPFSSIN 引脚，以启用串行外设数据写入。连续传输结束后，最后一位被捕获后一个 SSPCLKOUT 周期，SSPFSSOUT 引脚返回其空闲状态。

4.4.3.13. SPO=1, SPH=1 的 Motorola SPI 格式

图95展示了 SPO=1, SPH=1 的 Motorola SPI 格式的传输信号序列，涵盖单次和连续传输。

图95。SPO=1 和
SPH=1 的 Motorola S
PI 帧格式，支持
单次及连续传输



① 注意

图95中，Q 为未定义信号。

在此配置下，空闲期间：

- SSPCLKOUT 信号被强制置为高电平。
- SSPFSSOUT 信号被强制保持高电平
- 传输数据线 SSPTXD 被任意强制保持低电平
- nSSPOE 引脚使能信号被强制保持高电平（注意，该信号在 RP2040 中未连接至引脚）
- 当 PrimeCell SSP 配置为主设备时，nSSPCTLOE 线被拉低，启用 SSPCLKOUT 引脚，该使能信号为低有效
- 当 PrimeCell SSP 配置为从设备时，nSSPCTLOE 线被拉高，禁用 SSPCLKOUT 引脚，该使能信号为低有效。

若启用 PrimeCell SSP 且发送 FIFO 中存在有效数据，则传输起始由 SSPFSSOUT 主控信号被拉低表示。nSSPOE 线拉低，主设备 SSPTXD 输出端口被使能。再经过半个 SSPCLKOUT 周期后，主从设备的数据均被允许输出至各自传输线。同时，SSPCLKOUT 信号以下降沿转换方式被使能。数据随后在 SSPCLKOUT 信号的上升沿被采集，并于下降沿发出。

在所有位传输完成后，对于单字传输，SSPFSSOUT 线路将在最后一位被捕获后一个 SSPCLKOUT 周期恢复至空闲的高电平状态。

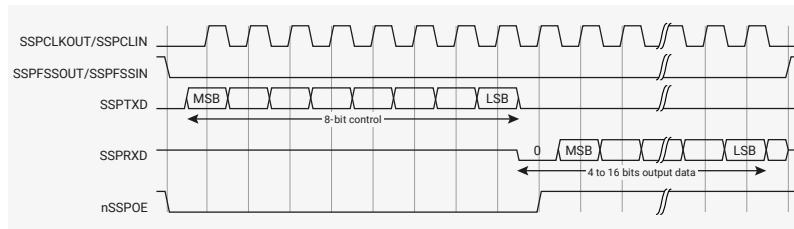
对于连续背靠背传输，SSPFSSOUT 引脚保持活动的低电平状态，直到最后一字的最后一一位被捕获，然后如前文所述返回至空闲状态。

对于连续背靠背传输，SSPFSSOUT 引脚在连续数据字之间保持低电平，终止方式与单字传输相同。

4.4.3.14. National Semiconductor Microwire 帧格式

图96展示了 National Semiconductor Microwire 的单帧格式。图97展示了连续帧传输时的相同格式。

Figure 96. Microwire frame format, single transfer



Microwire format is very similar to SPI format, except that transmission is half-duplex instead of full-duplex, using a master-slave message passing technique. Each serial transmission begins with an 8-bit control word that is transmitted from the PrimeCell SSP to the off-chip slave device. During this transmission, the PrimeCell SSP receives no incoming data. After the message has been sent, the off-chip slave decodes it and, after waiting one serial clock after the last bit of the 8-bit control message has been sent, responds with the required data. The returned data is 4 to 16 bits in length, making the total frame length in the range 13-25 bits.

In this configuration, during idle periods:

- SSPCLKOUT is forced LOW
- SSPFSSOUT is forced HIGH
- the transmit data line, SSPTXD, is arbitrarily forced LOW
- the nSSPOE pad enable signal is forced HIGH (note this is not connected to the pad in RP2040)

A transmission is triggered by writing a control byte to the transmit FIFO. The falling edge of SSPFSSOUT causes the value contained in the bottom entry of the transmit FIFO to be transferred to the serial shift register of the transmit logic, and the MSB of the 8-bit control frame to be shifted out onto the SSPTXD pin. SSPFSSOUT remains LOW for the duration of the frame transmission. The SSPRXD pin remains tristated during this transmission.

The off-chip serial slave device latches each control bit into its serial shifter on the rising edge of each SSPCLKOUT. After the last bit is latched by the slave device, the control byte is decoded during a one clock wait-state, and the slave responds by transmitting data back to the PrimeCell SSP. Each bit is driven onto SSPRXD line on the falling edge of SSPCLKOUT. The PrimeCell SSP in turn latches each bit on the rising edge of SSPCLKOUT. At the end of the frame, for single transfers, the SSPFSSOUT signal is pulled HIGH one clock period after the last bit has been latched in the receive serial shifter, that causes the data to be transferred to the receive FIFO.

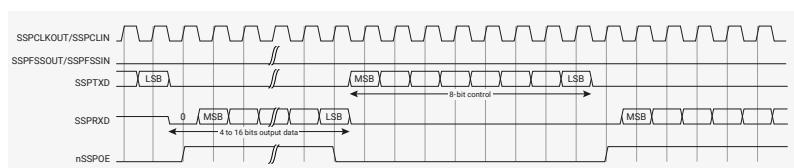
NOTE

The off-chip slave device can tristate the receive line either on the falling edge of SSPCLKOUT after the LSB has been latched by the receive shifter, or when the SSPFSSOUT pin goes HIGH.

For continuous transfers, data transmission begins and ends in the same manner as a single transfer. However, the SSPFSSOUT line is continuously asserted, held LOW, and transmission of data occurs back-to-back. The control byte of the next frame follows directly after the LSB of the received data from the current frame. Each of the received values is transferred from the receive shifter on the falling edge SSPCLKOUT, after the LSB of the frame has been latched into the PrimeCell SSP.

Figure 97 shows the National Semiconductor Microwire frame format when back-to-back frames are transmitted.

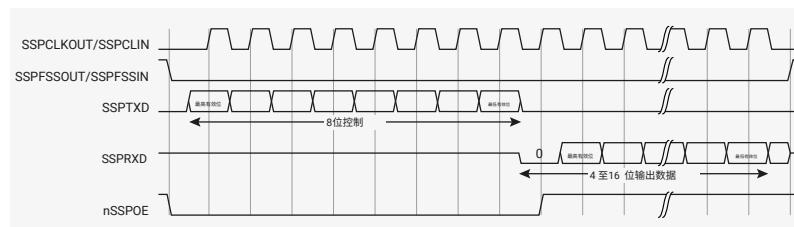
Figure 97. Microwire frame format, continuous transfers



In Microwire mode, the PrimeCell SSP slave samples the first bit of receive data on the rising edge of SSPCLKIN after SSPFSSIN has gone LOW. Masters that drive a free-running SSPCLKIN must ensure that the SSPFSSIN signal has sufficient setup and hold margins with respect to the rising edge of SSPCLKIN.

Figure 98 shows these setup and hold time requirements.

图96。Microwire
帧格式，单次
传输



Microwire格式与SPI格式极为相似，不同之处在于传输采用半双工而非全双工，使用主从消息传递技术。每次串行传输均始于一个由PrimeCell SSP发送至片外从设备的8位控制字。在该传输过程中，PrimeCell SSP不会接收任何输入数据。消息发送完成后，片外从设备对其进行解码，并在最后一位8位控制消息发送完成后等待一个串行时钟周期，然后返回所需数据。返回的数据长度为4至16位，使得总帧长度在13至25位之间。

在此配置下，空闲期间：

- SSPCLKOUT被强制为低电平
- SSPFSSOUT被强制为高电平
- 发送数据线SSPTXD被任意强制为低电平
- nSSPOE引脚使能信号被强制保持高电平（注意，该信号在RP2040中未连接至引脚）

通过向发送FIFO写入控制字节来触发传输。SSPFSSOUT的下降沿促使发送FIFO底部的数据被传输至发送逻辑的串行移位寄存器，且8位控制帧的最高位（MSB）被移出至SSPTXD引脚。在整个帧传输期间，SSPFSSOUT保持低电平。在此传输期间，SSPRXD引脚保持三态（高阻）状态。

片外串行从设备在每个SSPCLKOUT上升沿将每个控制位锁存至其串行移位寄存器中。

在从设备锁存最后一位后，控制字节将在一个时钟等待周期内被译码，从设备随后通过传输数据响应PrimeCell SSP。每个位于SSPCLKOUT下降沿驱动至SSPRXD线，PrimeCell SSP则于SSPCLKOUT上升沿锁存每个位。帧结束时，对于单次传输，SSPFSSOUT信号在接收串行移位寄存器锁存最后一位后一个时钟周期被拉高，促使数据传输至接收FIFO。

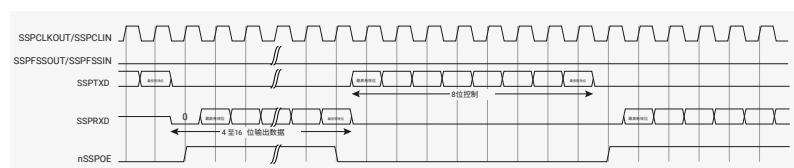
注意

片外设备可在接收移位器锁存最低有效位后的SSPCLKOUT下降沿，或SSPFSSOUT引脚变为高电平时，将接收线置于三态（高阻）状态。

对于连续传输，数据传输的开始和结束方式与单次传输相同。然而，SSPFSSOUT线持续断言，保持低电平，且数据连续传输。下一帧的控制字节紧随当前帧接收数据的最低有效位之后。每个接收值均在SSPCLKOUT下降沿由接收移位器传输，前提是该帧的最低有效位已锁存至PrimeCell SSP。

图97展示了在连续传输多帧时，National Semiconductor Microwire 的帧格式。

图97。Microwire
帧格式，
连续传输



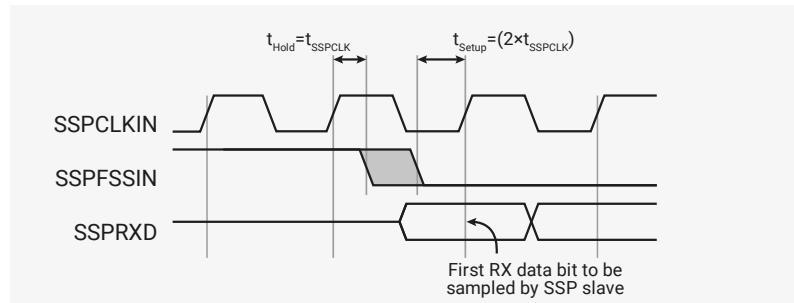
在Microwire模式下，PrimeCell SSP从属设备在SSPFSSIN拉低后，于SSPCLKIN上升沿采样接收数据的第一个位。驱动自由运行SSPCLKIN的主设备必须确保SSPFSSIN信号相对于SSPCLKIN上升沿具备充分的建立时间和保持时间裕量。

图98显示了这些建立时间和保持时间的要求。

With respect to the SSPCLKIN rising edge on which the first bit of receive data is to be sampled by the PrimeCell SSP slave, SSPFSSIN must have a setup of at least two times the period of SSPCLK on which the PrimeCell SSP operates.

With respect to the SSPCLKIN rising edge previous to this edge, SSPFSSIN must have a hold of at least one SSPCLK period.

Figure 98. Microwire frame format, SSPFSSIN input setup and hold requirements



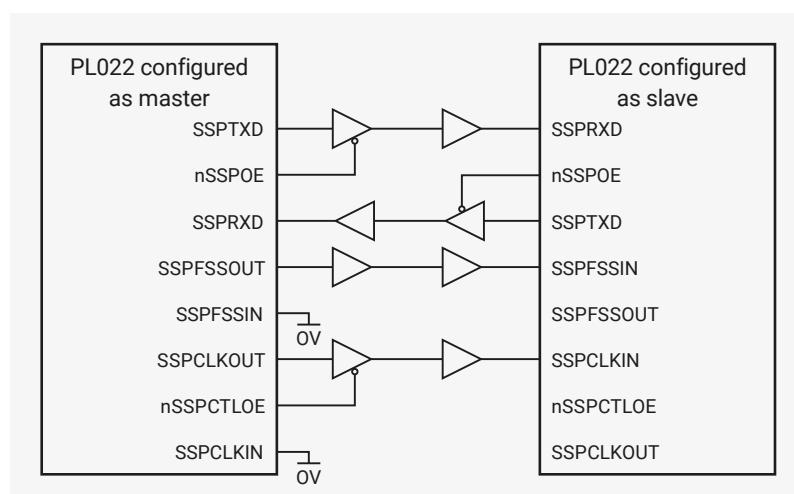
4.4.3.15. Examples of master and slave configurations

[Figure 99](#), [Figure 100](#), and [Figure 101](#) shows how you can connect the PrimeCell SSP (PL022) peripheral to other synchronous serial peripherals, when it is configured as a master or a slave.

NOTE

The SSP (PL022) does not support dynamic switching between master and slave in a system. Each instance is configured and connected either as a master or slave.

Figure 99. PrimeCell SSP master coupled to a PL022 slave

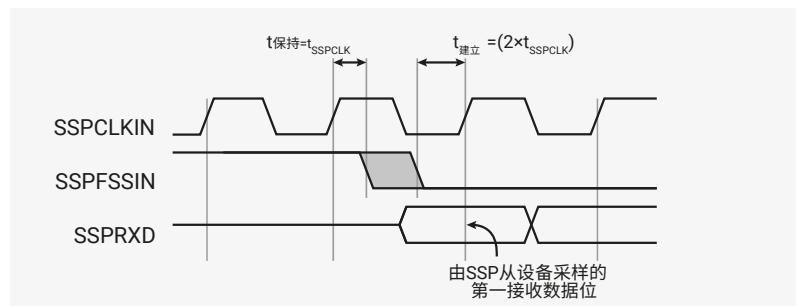


[Figure 100](#) shows an PrimeCell SSP (PL022), configured as master, interfaces to a Motorola SPI slave. The SPI Slave Select (SS) signal is permanently tied LOW and configures it as a slave. Similar to the above operation, the master can broadcast to the slave through the master PrimeCell SSP SSPTXD line. In response, the slave drives its SPI MISO port onto the SSPrXD line of the master.

关于PrimeCell SSP从设备采样接收数据的第一位时所对应的SSPCLKIN上升沿，SSPFSSIN必须具有至少两倍于PrimeCell SSP运行时所用SSPCLK周期的建立时间。

关于该上升沿之前的SSPCLKIN上升沿，SSPFSSIN必须保持至少一个SSPCLK周期的保持时间。

图98。Microwire
帧格式，SS
PFSSIN输入的建立
和保持时间要求



4.4.3.15. 主从设备配置示例

图99、图100和图101展示了当PrimeCell SSP（PL022）外设配置为主设备或从设备时，如何连接其他同步串行外设。

注意

SSP（PL022）不支持系统中主从角色的动态切换。每个实例均被配置并连接为主设备或从设备。

图99. PrimeCell
SSP主设备连接至
PL022从设备

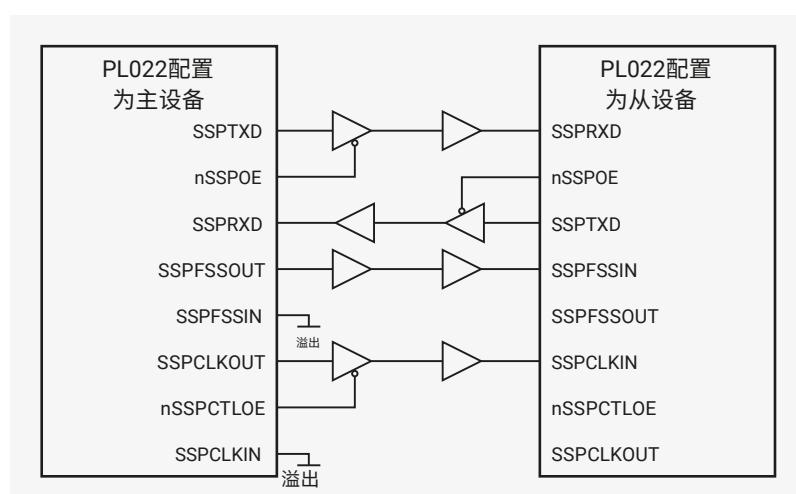


图100展示了配置为主设备的PrimeCell SSP（PL022）如何与摩托罗拉SPI从设备接口。SPI从设备选择（SS）信号被永久拉低，用于将其配置为从设备。与上述操作类似，主设备可通过主设备PrimeCell SSP的SSPTXD线向从设备广播。作为响应，从设备将其SPI MISO端口驱动至主设备的SSPRXD线上。

Figure 100. PrimeCell SSP master coupled to an SPI slave

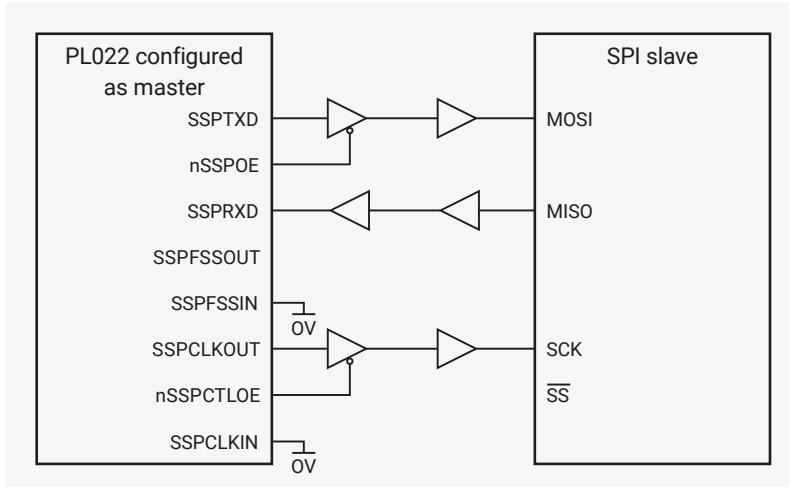
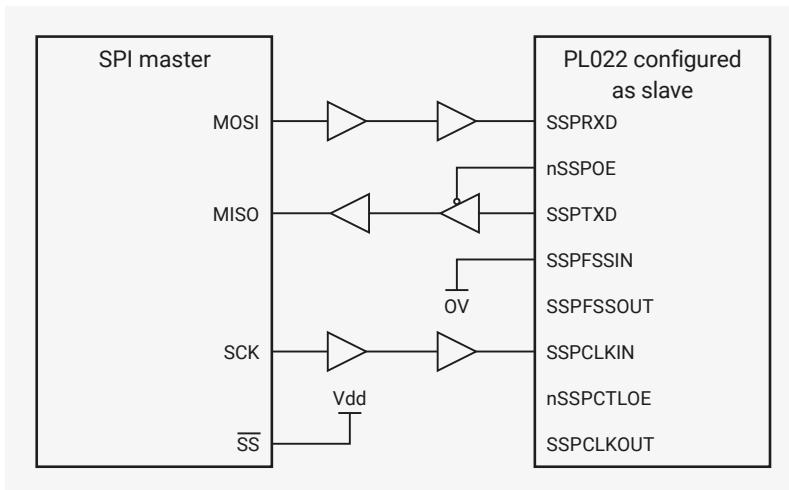


Figure 101 shows a Motorola SPI configured as a master and interfaced to an instance of a PrimeCell SSP (PL022) configured as a slave. In this case, the slave Select Signal (SS) is permanently tied HIGH to configure it as a master. The master can broadcast to the slave through the master SPI MOSI line and in response, the slave drives its nSSPOE signal LOW. This enables its SSPTXD data onto the MISO line of the master.

Figure 101. SPI master coupled to a PrimeCell SSP slave



4.4.3.16. PrimeCell DMA interface

The PrimeCell SSP provides an interface to connect to the DMA controller. The PrimeCell SSP DMA control register, SSPDMACR controls the DMA operation of the PrimeCell SSP.

The DMA interface includes the following signals, for receive:

SSPRXDMASREQ

Single-character DMA transfer request, asserted by the SSP. This signal is asserted when the receive FIFO contains at least one character.

SSPRXDMABREQ

Burst DMA transfer request, asserted by the SSP. This signal is asserted when the receive FIFO contains four or more characters.

SSPRXDMACLR

DMA request clear, asserted by the DMA controller to clear the receive request signals. If DMA burst transfer is requested, the clear signal is asserted during the transfer of the last data in the burst.

The DMA interface includes the following signals, for transmit:

图 100。PrimeCell
SSP 主设备与 SPI 从
设备的连接

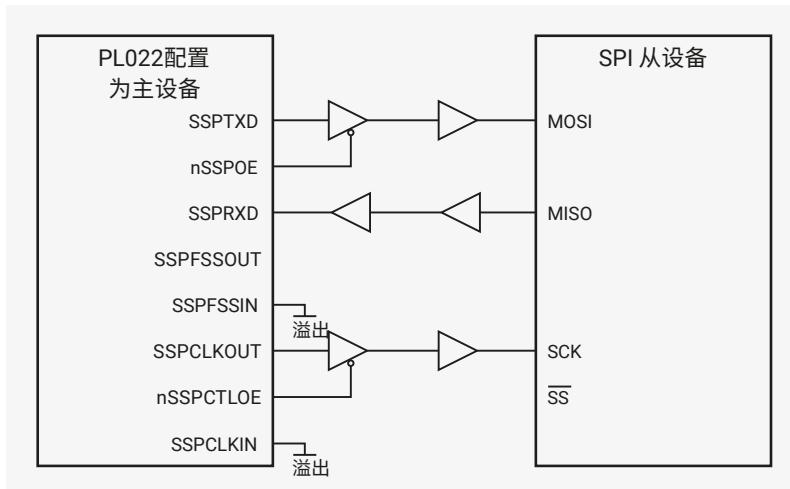
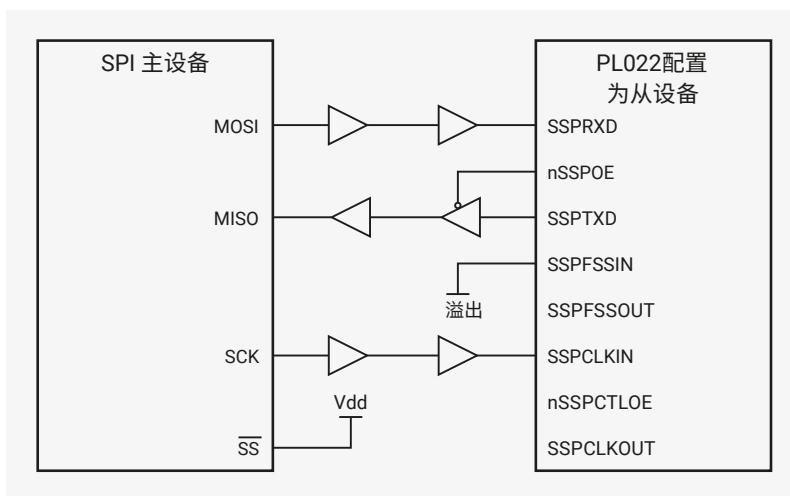


图 101 显示了一台配置为主设备的 Motorola SPI，连接至配置为从设备的 PrimeCell SSP（PL022）实例。在此情况下，从设备的选择信号（SS）被永久拉高，以将其配置为主设备。主设备可通过主设备 SPI MOSI 线向从设备广播，作为响应，从设备将其 nSSPOE 信号拉低，从而使其 SSPTXD 数据能够输出至主设备的 MISO 线上。

图 101。SPI 主控
连接至 PrimeCell
SSP 从属设备



4.4.3.16. PrimeCell DMA 接口

PrimeCell SSP 提供连接至 DMA 控制器的接口。PrimeCell SSP 的 DMA 控制寄存器 SSPDMACR 用于控制 PrimeCell SSP 的 DMA 操作。

DMA 接口包括以下接收信号：

SSPRXDMASREQ

单字符 DMA 传输请求，由 SSP 断言。当接收 FIFO 中至少有一个字符时，该信号被断言。

SSPRXDMABREQ

突发 DMA 传输请求，由 SSP 断言。当接收 FIFO 中包含四个或更多字符时，该信号被断言。

SSPRXDMACLR

DMA 请求清除，由 DMA 控制器断言，用以清除接收请求信号。当请求 DMA 突发传输时，清除信号在突发传输的最后一个数据传输期间断言。

DMA 接口包括以下发送信号：

SSPTXDMASREQ

Single-character DMA transfer request, asserted by the SSP. This signal is asserted when there is at least one empty location in the transmit FIFO.

SSPTXDMABREQ

Burst DMA transfer request, asserted by the SSP. This signal is asserted when the transmit FIFO contains four characters or fewer.

SSPTXDMACLR

DMA request clear, asserted by the DMA controller, to clear the transmit request signals. If a DMA burst transfer is requested, the clear signal is asserted during the transfer of the last data in the burst.

The burst transfer and single transfer request signals are not mutually exclusive. They can both be asserted at the same time. For example, when there is more data than the watermark level of four in the receive FIFO, the burst transfer request, and the single transfer request, are asserted. When the amount of data left in the receive FIFO is less than the watermark level, the single request only is asserted. This is useful for situations where the number of characters left to be received in the stream is less than a burst.

For example, if 19 characters must be received, the DMA controller then transfers four bursts of four characters, and three single transfers to complete the stream.

NOTE

For the remaining three characters, the PrimeCell SSP does not assert the burst request.

Each request signal remains asserted until the relevant DMA clear signal is asserted. After the request clear signal is deasserted, a request signal can become active again, depending on the conditions that previous sections describe. All request signals are deasserted if the PrimeCell SSP is disabled, or the DMA enable signal is cleared.

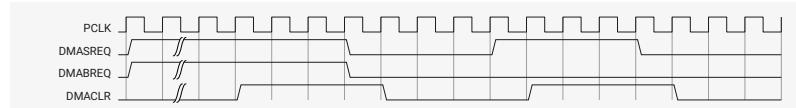
Table 495 shows the trigger points for DMABREQ, for both the transmit and receive FIFOs.

Table 495. DMA trigger points for the transmit and receive FIFOs

Burst length		
Watermark level	Transmit, number of empty locations	Receive, number of filled locations
1/2	4	4

Figure 102 shows the timing diagram for both a single transfer request, and a burst transfer request, with the appropriate DMA clear signal. The signals are all synchronous to PCLK.

Figure 102. DMA transfer waveforms



4.4.4. List of Registers

The SPI0 and SPI1 registers start at base addresses of [0x4003c000](#) and [0x40040000](#) respectively (defined as [SPI0_BASE](#) and [SPI1_BASE](#) in SDK).

Table 496. List of SPI registers

Offset	Name	Info
0x000	SSPCR0	Control register 0, SSPCR0 on page 3-4
0x004	SSPCR1	Control register 1, SSPCR1 on page 3-5
0x008	SSPDR	Data register, SSPDR on page 3-6
0x00c	SSPSR	Status register, SSPSR on page 3-7
0x010	SSPCPSR	Clock prescale register, SSPCPSR on page 3-8

SSPTXDMASREQ

单字符 DMA 传输请求，由 SSP 断言。当发送 FIFO 中至少有一个空位时，该信号被断言。

SSPTXDMABREQ

突发 DMA 传输请求，由 SSP 断言。当发送 FIFO 中包含四个或更少字符时，该信号被断言。

SSPTXDMACLR

DMA 请求清除信号，由 DMA 控制器断言，用于清除传输请求信号。若请求 DMA 突发传输，则在传输突发的最后一笔数据时断言清除信号。

突发传输请求信号与单次传输请求信号并非互斥。二者可以同时断言。例如，当接收 FIFO 中的数据量超过四的水位线时，突发传输请求和单次传输请求会被断言。当接收 FIFO 中剩余数据量低于水位线时，仅断言单次传输请求。该设置适用于流中剩余待接收字符数少于一个突发长度的情况。

例如，若需接收 19 个字符，DMA 控制器先传输四个 4 字符的突发，随后进行 3 次单次传输以完成数据流。

注意

对于剩余的 3 个字符，PrimeCell SSP 不会断言突发请求信号。

每个请求信号将持续断言，直到相应的 DMA 清除信号被断言。在请求清除信号撤销后，请求信号可再次变为有效，具体取决于前述章节所述的条件。如果 PrimeCell SSP 被禁用，或者 DMA 使能信号被清除，则所有请求信号均被撤销。

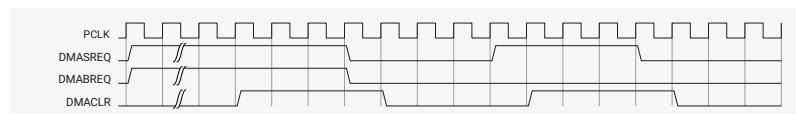
[表 495 显示了 DMABREQ 对于发送和接收 FIFO 的触发点。](#)

表 495。发送与接收 FIFO 的 DMA 触发点

突发长度		
水位线	发送端，空闲位置数量	接收端，已占用位置数量
1/2	4	4

[图 102 展示了在适用的 DMA 清除信号下，单次传输请求和突发传输请求的时序图。所有信号均与 PCLK 同步。](#)

图 102。DMA 传输波形



4.4.4. 寄存器列表

SPI0 和 SPI1 寄存器的基址分别为 [0x4003c000](#) 和 [0x40040000](#)（在 SDK 中定义为 SPI0_BASE 和 SPI1_BASE）。

表 496. SPI 寄存器列表

偏移量	名称	说明
0x000	SSPCR0	控制寄存器 0 SSPCR0，详见第 3-4 页
0x004	SSPCR1	控制寄存器 1 SSPCR1，详见第 3-5 页
0x008	SSPDR	数据寄存器 SSPDR，详见第 3-6 页
0x00c	SSPSR	状态寄存器 SSPSR，详见第 3-7 页
0x010	SSPCPSR	时钟预分频寄存器 SSPCPSR，详见第 3-8 页

Offset	Name	Info
0x014	SSPIMSC	Interrupt mask set or clear register, SSPIMSC on page 3-9
0x018	SSPRIS	Raw interrupt status register, SSPRIS on page 3-10
0x01c	SSPMIS	Masked interrupt status register, SSPMIS on page 3-11
0x020	SSPICR	Interrupt clear register, SSPICR on page 3-11
0x024	SSPDMACR	DMA control register, SSPDMACR on page 3-12
0xfe0	SSPPERIPHIDO	Peripheral identification registers, SSPPERIPHID0-3 on page 3-13
0xfe4	SSPPERIPHID1	Peripheral identification registers, SSPPERIPHID0-3 on page 3-13
0xfe8	SSPPERIPHID2	Peripheral identification registers, SSPPERIPHID0-3 on page 3-13
0xfec	SSPPERIPHID3	Peripheral identification registers, SSPPERIPHID0-3 on page 3-13
0xff0	SSPPCELLID0	PrimeCell identification registers, SSPPCELLID0-3 on page 3-16
0xff4	SSPPCELLID1	PrimeCell identification registers, SSPPCELLID0-3 on page 3-16
0xff8	SSPPCELLID2	PrimeCell identification registers, SSPPCELLID0-3 on page 3-16
0ffc	SSPPCELLID3	PrimeCell identification registers, SSPPCELLID0-3 on page 3-16

SPI: SSPCR0 Register

Offset: 0x000

Description

Control register 0, SSPCR0 on page 3-4

Table 497. SSPCR0 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:8	SCR: Serial clock rate. The value SCR is used to generate the transmit and receive bit rate of the PrimeCell SSP. The bit rate is: F SSPCLK CPSDVSR x (1+SCR) where CPSDVSR is an even value from 2-254, programmed through the SSPCPSR register and SCR is a value from 0-255.	RW	0x00
7	SPH: SSPCLKOUT phase, applicable to Motorola SPI frame format only. See Motorola SPI frame format on page 2-10.	RW	0x0
6	SPO: SSPCLKOUT polarity, applicable to Motorola SPI frame format only. See Motorola SPI frame format on page 2-10.	RW	0x0
5:4	FRF: Frame format: 00 Motorola SPI frame format. 01 TI synchronous serial frame format. 10 National Microwire frame format. 11 Reserved, undefined operation.	RW	0x0
3:0	DSS: Data Size Select: 0000 Reserved, undefined operation. 0001 Reserved, undefined operation. 0010 Reserved, undefined operation. 0011 4-bit data. 0100 5-bit data. 0101 6-bit data. 0110 7-bit data. 0111 8-bit data. 1000 9-bit data. 1001 10-bit data. 1010 11-bit data. 1011 12-bit data. 1100 13-bit data. 1101 14-bit data. 1110 15-bit data. 1111 16-bit data.	RW	0x0

SPI: SSPCR1 Register

Offset: 0x004

Description

Control register 1, SSPCR1 on page 3-5

偏移量	名称	说明
0x014	SSPIMSC	中断屏蔽设置/清除寄存器 SSPIMSC，详见第3-9页
0x018	SSPRIS	原始中断状态寄存器 SSPRIS，详见第3-10页
0x01c	SSPMIS	屏蔽中断状态寄存器 SSPMIS，详见第3-11页
0x020	SSPICR	中断清除寄存器，SSPICR，见第3-11页
0x024	SSPDMACR	DMA控制寄存器，SSPDMACR，见第3-12页
0xfe0	SSPPERIPHIDO	外设标识寄存器，SSPPERIPHIDO-3，见第3-13页
0xfe4	SSPPERIPHID1	外设标识寄存器，SSPPERIPHID1-3，见第3-13页
0xfe8	SSPPERIPHID2	外设标识寄存器，SSPPERIPHID2-3，见第3-13页
0xfec	SSPPERIPHID3	外设标识寄存器，SSPPERIPHID3-3，见第3-13页
0xff0	SSPPCELLID0	PrimeCell标识寄存器，SSPPCELLID0-3，见第3-16页
0xff4	SSPPCELLID1	PrimeCell标识寄存器，SSPPCELLID1-3，见第3-16页
0xff8	SSPPCELLID2	PrimeCell标识寄存器，SSPPCELLID2-3，见第3-16页
0ffc	SSPPCELLID3	PrimeCell标识寄存器，SSPPCELLID3-3，见第3-16页

SPI: SSPCR0 寄存器

偏移：0x000

描述

控制寄存器0 SSPCR0，详见第3-4页

表 497. SSPCR0
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:8	SCR : 串行时钟速率。SCR值用于生成PrimeCell SSP的发送与接收比特率。比特率计算公式为： $F_{SSPCLK} \div (CPSDVSR \times (1+SCR))$ ，其中CPSDVSR为通过SSPCPSR寄存器配置的2至254之间的偶数，SCR则为0至255之间的值。	读写	0x00
7	SPH : SSPCLKOUT 相位，仅适用于 Motorola SPI 帧格式。请参见第 2-10 页的 Motorola SPI 帧格式说明。	读写	0x0
6	SPO : SSPCLKOUT 极性，仅适用于 Motorola SPI 帧格式。请参见第 2-10 页的 Motorola SPI 帧格式说明。	读写	0x0
5:4	FRF : 帧格式：00 Motorola SPI 帧格式；01 TI 同步串行帧格式；10 National Semiconductor Microwire 帧格式；11 保留，未定义操作。	读写	0x0
3:0	DSS : 数据位选择：0000 保留，未定义操作；0001 保留，未定义操作；0010 保留，未定义操作；0011 4 位数据。0100 5 位数据；0101 6 位数据；0110 7 位数据；0111 8 位数据；1000 9 位数据；1001 10 位数据；1010 11 位数据；1011 12 位数据；1100 13 位数据。1101 14 位数据；1110 15 位数据；1111 16 位数据。	读写	0x0

SPI: SSPCR1 寄存器

偏移：0x004

描述

控制寄存器1 SSPCR1，详见第3-5页

Table 498. SSPCR1 Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	SOD : Slave-mode output disable. This bit is relevant only in the slave mode, MS=1. In multiple-slave systems, it is possible for an PrimeCell SSP master to broadcast a message to all slaves in the system while ensuring that only one slave drives data onto its serial output line. In such systems the RXD lines from multiple slaves could be tied together. To operate in such systems, the SOD bit can be set if the PrimeCell SSP slave is not supposed to drive the SSPTXD line: 0 SSP can drive the SSPTXD output in slave mode. 1 SSP must not drive the SSPTXD output in slave mode.	RW	0x0
2	MS : Master or slave mode select. This bit can be modified only when the PrimeCell SSP is disabled, SSE=0: 0 Device configured as master, default. 1 Device configured as slave.	RW	0x0
1	SSE : Synchronous serial port enable: 0 SSP operation disabled. 1 SSP operation enabled.	RW	0x0
0	LBM : Loop back mode: 0 Normal serial port operation enabled. 1 Output of transmit serial shifter is connected to input of receive serial shifter internally.	RW	0x0

SPI: SSPDR Register

Offset: 0x008

Description

Data register, SSPDR on page 3-6

Table 499. SSPDR Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	DATA : Transmit/Receive FIFO: Read Receive FIFO. Write Transmit FIFO. You must right-justify data when the PrimeCell SSP is programmed for a data size that is less than 16 bits. Unused bits at the top are ignored by transmit logic. The receive logic automatically right-justifies.	RWF	-

SPI: SSPSR Register

Offset: 0x00c

Description

Status register, SSPSR on page 3-7

Table 500. SSPSR Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4	BSY : PrimeCell SSP busy flag, RO: 0 SSP is idle. 1 SSP is currently transmitting and/or receiving a frame or the transmit FIFO is not empty.	RO	0x0
3	RFF : Receive FIFO full, RO: 0 Receive FIFO is not full. 1 Receive FIFO is full.	RO	0x0
2	RNE : Receive FIFO not empty, RO: 0 Receive FIFO is empty. 1 Receive FIFO is not empty.	RO	0x0
1	TNF : Transmit FIFO not full, RO: 0 Transmit FIFO is full. 1 Transmit FIFO is full.	RO	0x1
0	TFE : Transmit FIFO empty, RO: 0 Transmit FIFO is not empty. 1 Transmit FIFO is empty.	RO	0x1

表 498. SSPCR1
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	SOD : 从机模式输出禁用。此位仅在从机模式 (MS=1) 下有效。在多从机系统中, PrimeCell SSP 主设备可以向系统中所有从设备广播消息, 同时确保只有一个从设备驱动其串行输出线路的数据。在此类系统中, 多个从设备的 RXD 线路可能被并联连接。为保证在此类系统中正常工作, 如 PrimeCell SSP 从设备不应驱动 SSPTXD 线路, 则可设置 SOD 位: 0 表示 SSP 在从机模式下可驱动 SSPTXD 输出; 1 表示 SSP 在从机模式下不得驱动 SSPTXD 输出。	读写	0x0
2	MS : 主从模式选择。该位仅可在 PrimeCell SSP 禁用时修改 (SSE=0) : 0 表示设备配置为主设备, 默认值; 1 表示设备配置为从设备。	读写	0x0
1	SSE : 同步串行端口使能: 0 表示禁用 SSP 操作。1 表示启用 SSP 操作。	读写	0x0
0	LBM : 环回模式: 0 启用正常串行端口操作。1 传输串行移位寄存器的输出在内部连接至接收串行移位寄存器的输入。	读写	0x0

SPI: SSPDR 寄存器

偏移: 0x008

说明

数据寄存器 SSPDR, 详见第3-6页

表 499. SSPDR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	DATA : 传输/接收 FIFO: 读取接收 FIFO, 写入传输 FIFO。当 PrimeCell SSP 被配置为小于 16 位的数据大小时, 数据必须右对齐。传输逻辑将忽略高位未使用的位。 接收逻辑会自动进行右对齐。	RWF	-

SPI: SSPSR 寄存器

偏移: 0x00c

说明

状态寄存器 SSPSR, 详见第3-7页

表 500. SSPSR
寄存器

位	描述	类型	复位值
31:5	保留。	-	-
4	BSY : PrimeCell SSP 忙标志, RO: 0 表示 SSP 空闲。1 表示 SSP 正在发送和/或接收一帧, 或传输 FIFO 非空。	只读	0x0
3	RFF : 接收FIFO满, RO: 0 表示接收FIFO未满, 1 表示接收FIFO已满。	只读	0x0
2	RNE : 接收FIFO非空, RO: 0 表示接收FIFO为空, 1 表示接收FIFO非空。	只读	0x0
1	TNF : 发送FIFO未满, RO: 0 表示发送FIFO已满, 1 表示发送FIFO未满。	只读	0x1
0	TFE : 发送FIFO空, RO: 0 表示发送FIFO非空, 1 表示发送FIFO为空。	只读	0x1

SPI: SSPCPSR Register

Offset: 0x010

Description

Clock prescale register, SSPCPSR on page 3-8

Table 501. SSPCPSR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	CPSDVSR: Clock prescale divisor. Must be an even number from 2-254, depending on the frequency of SSPCLK. The least significant bit always returns zero on reads.	RW	0x00

SPI: SSPIMSC Register

Offset: 0x014

Description

Interrupt mask set or clear register, SSPIMSC on page 3-9

Table 502. SSPIMSC Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	TXIM: Transmit FIFO interrupt mask: 0 Transmit FIFO half empty or less condition interrupt is masked. 1 Transmit FIFO half empty or less condition interrupt is not masked.	RW	0x0
2	RXIM: Receive FIFO interrupt mask: 0 Receive FIFO half full or less condition interrupt is masked. 1 Receive FIFO half full or less condition interrupt is not masked.	RW	0x0
1	RTIM: Receive timeout interrupt mask: 0 Receive FIFO not empty and no read prior to timeout period interrupt is masked. 1 Receive FIFO not empty and no read prior to timeout period interrupt is not masked.	RW	0x0
0	RORIM: Receive overrun interrupt mask: 0 Receive FIFO written to while full condition interrupt is masked. 1 Receive FIFO written to while full condition interrupt is not masked.	RW	0x0

SPI: SSPRIS Register

Offset: 0x018

Description

Raw interrupt status register, SSPRIS on page 3-10

Table 503. SSPRIS Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	TXRIS: Gives the raw interrupt state, prior to masking, of the SSPTXINTR interrupt	RO	0x1
2	RXRIS: Gives the raw interrupt state, prior to masking, of the SSPRXINTR interrupt	RO	0x0
1	RTRIS: Gives the raw interrupt state, prior to masking, of the SSPRTINTR interrupt	RO	0x0

SPI: SSPCPSR 寄存器

偏移: 0x010

描述

时钟预分频寄存器 SSPCPSR，详见第3-8页

表 501. SSPCPSR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	CPSDVS : 时钟预分频除数。必须为2至254之间的偶数，具体取决于SS PCLK的频率。读取时最低有效位始终返回零。	读写	0x00

SPI: SSPIMSC 寄存器

偏移: 0x014

描述

中断屏蔽设置/清除寄存器 SSPIMSC，详见第3-9页

表 502. SSPIMSC
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXIM : 发送FIFO中断屏蔽：0 表示发送FIFO半空或更少状态中断被屏蔽，1 表示发送FIFO半空或更少状态中断未被屏蔽。	读写	0x0
2	RXIM : 接收FIFO中断屏蔽：0 接收FIFO半满或以下条件中断被屏蔽。1 接收FIFO半满或以下条件中断未被屏蔽。	读写	0x0
1	RTIM : 接收超时中断屏蔽：0 接收FIFO非空且超时期间未读取的中断被屏蔽。1 接收FIFO非空且超时期间未读取的中断未被屏蔽。	读写	0x0
0	RORIM : 接收溢出中断屏蔽：0 接收FIFO满时写入的中断被屏蔽。1 接收FIFO满时写入的中断未被屏蔽。	读写	0x0

SPI: SSPRIS寄存器

偏移量: 0x018

描述

原始中断状态寄存器 SSPRIS，详见第3-10页

表 503. SSPRIS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXRIS : 反映SSPTXINTR中断的原始状态（未屏蔽）	只读	0x1
2	RXRIS : 反映SSPRXINTR中断的原始状态（未屏蔽）	只读	0x0
1	RTRIS : 反映SSPRTINTR中断的原始状态（未屏蔽）	只读	0x0

Bits	Description	Type	Reset
0	RORRIS: Gives the raw interrupt state, prior to masking, of the SSPRORINTR interrupt	RO	0x0

SPI: SSPMIS Register

Offset: 0x01c

Description

Masked interrupt status register, SSPMIS on page 3-11

Table 504. SSPMIS Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	TXMIS: Gives the transmit FIFO masked interrupt state, after masking, of the SSPTXINTR interrupt	RO	0x0
2	RXMIS: Gives the receive FIFO masked interrupt state, after masking, of the SSPRXINTR interrupt	RO	0x0
1	RTMIS: Gives the receive timeout masked interrupt state, after masking, of the SSPRTINTR interrupt	RO	0x0
0	RORMIS: Gives the receive over run masked interrupt status, after masking, of the SSPRORINTR interrupt	RO	0x0

SPI: SSPICR Register

Offset: 0x020

Description

Interrupt clear register, SSPICR on page 3-11

Table 505. SSPICR Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	RTIC: Clears the SSPRTINTR interrupt	WC	0x0
0	RORIC: Clears the SSPRORINTR interrupt	WC	0x0

SPI: SSPDMACR Register

Offset: 0x024

Description

DMA control register, SSPDMACR on page 3-12

Table 506. SSPDMACR Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	TXDMAE: Transmit DMA Enable. If this bit is set to 1, DMA for the transmit FIFO is enabled.	RW	0x0
0	RXDMAE: Receive DMA Enable. If this bit is set to 1, DMA for the receive FIFO is enabled.	RW	0x0

SPI: SSPPERIPHID0 Register

Offset: 0xfe0

位	描述	类型	复位值
0	RORRIS : 提供SSPRORINTR中断在屏蔽前的原始中断状态	只读	0x0

SPI: SSPMIS寄存器

偏移: 0x01c

说明

屏蔽中断状态寄存器 SSPMIS, 详见第3-11页

表504. SSPMIS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	TXMIS : 提供SSPTXINTR中断在屏蔽后的发送FIFO屏蔽中断状态 SSPTXINTR中断	只读	0x0
2	RXMIS : 提供SSPRXINTR中断在屏蔽后的接收FIFO屏蔽中断状态	只读	0x0
1	RTMIS : 提供SSPRTINTR中断在屏蔽后的接收超时屏蔽中断状态	只读	0x0
0	RORMIS : 提供SSPRORINTR中断在屏蔽后的接收溢出屏蔽中断状态	只读	0x0

SPI: SSPICR寄存器

偏移: 0x020

说明

中断清除寄存器, SSPICR, 见第3-11页

表505. SSPICR
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	RTIC : 清除SSPRTINTR中断	WC	0x0
0	RORIC : 清除SSPRORINTR中断	WC	0x0

SPI: SSPDMACR寄存器

偏移: 0x024

描述

DMA控制寄存器, SSPDMACR, 见第3-12页

表 506. SSPDMACR
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TXDMAE : 发送 DMA 启用。若该位设置为 1，则启用发送 FIFO 的 DMA。	读写	0x0
0	RXDMAE : 接收 DMA 启用。若该位设置为 1，则启用接收 FIFO 的 DMA。	读写	0x0

SPI: SSPPERIPHIDO 寄存器

偏移量: 0xfe0

Description

Peripheral identification registers, SSPPeriphID0-3 on page 3-13

Table 507.
SSPPERIPHID0
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	PARTNUMBER0: These bits read back as 0x22	RO	0x22

SPI: SSPPERIPHID1 Register

Offset: 0xfe4

Description

Peripheral identification registers, SSPPeriphID0-3 on page 3-13

Table 508.
SSPPERIPHID1
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:4	DESIGNERO: These bits read back as 0x1	RO	0x1
3:0	PARTNUMBER1: These bits read back as 0x0	RO	0x0

SPI: SSPPERIPHID2 Register

Offset: 0xfe8

Description

Peripheral identification registers, SSPPeriphID0-3 on page 3-13

Table 509.
SSPPERIPHID2
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:4	REVISION: These bits return the peripheral revision	RO	0x3
3:0	DESIGNER1: These bits read back as 0x4	RO	0x4

SPI: SSPPERIPHID3 Register

Offset: 0xfec

Description

Peripheral identification registers, SSPPeriphID0-3 on page 3-13

Table 510.
SSPPERIPHID3
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	CONFIGURATION: These bits read back as 0x00	RO	0x00

SPI: SSPPCELLID0 Register

Offset: 0xff0

Description

PrimeCell identification registers, SSPPCellID0-3 on page 3-16

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 507
SSPPERIPHID
0 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	PARTNUMBER0 : 此位读取值为 0x22	只读	0x22

SPI: SSPPERIPHID1 寄存器

偏移: 0xfe4

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 508
SSPPERIPHID
1 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	DESIGNERO : 这些位的读出值为0x1	只读	0x1
3:0	PARTNUMBER1 : 这些位读取值为 0x0	只读	0x0

SPI: SSPPERIPHID2 寄存器

偏移: 0xfe8

描述

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 509
SSPPERIPHID
2 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:4	REVISION : 此位返回外设修订版本	只读	0x3
3:0	DESIGNER1 : 这些位读取值为 0x4	只读	0x4

SPI: SSPPERIPHID3 寄存器

偏移: 0fec

说明

外设标识寄存器，SSPPeriphID0-3，见第3-13页

表 510
SSPPERIPHID3
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	配置 : 这些位读取值为 0x00	只读	0x00

SPI: SSPPCELLID0 寄存器

偏移量: 0xff0

描述

PrimeCell标识寄存器，SSPPCellID0-3，见第3-16页

Table 511.
SSPPCELLID0 Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	SSPPCELLID0: These bits read back as 0x0D	RO	0x0d

SPI: SSPPCELLID1 Register

Offset: 0xff4

Description

PrimeCell identification registers, SSPPCellID0-3 on page 3-16

Table 512.
SSPPCELLID1 Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	SSPPCELLID1: These bits read back as 0xF0	RO	0xf0

SPI: SSPPCELLID2 Register

Offset: 0xff8

Description

PrimeCell identification registers, SSPPCellID0-3 on page 3-16

Table 513.
SSPPCELLID2 Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	SSPPCELLID2: These bits read back as 0x05	RO	0x05

SPI: SSPPCELLID3 Register

Offset: 0ffc

Description

PrimeCell identification registers, SSPPCellID0-3 on page 3-16

Table 514.
SSPPCELLID3 Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	SSPPCELLID3: These bits read back as 0xB1	RO	0xb1

4.5. PWM

4.5.1. Overview

Pulse width modulation (PWM) is a scheme where a digital signal provides a smoothly varying average voltage. This is achieved with positive pulses of some controlled width, at regular intervals. The fraction of time spent high is known as the duty cycle. This may be used to approximate an analog output, or control switchmode power electronics.

The RP2040 PWM block has 8 identical slices. Each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a total of up to 16 controllable PWM outputs. All 30 GPIO pins can be driven by the PWM block.

表 511
SSPPCELLID0 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID0 : 这些位读取值为 0x0D	只读	0x0d

SPI: SSPPCELLID1 寄存器

偏移: 0xff4

说明

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 512
SSPPCELLID1 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID1 : 这些位读取值为 0xF0	只读	0xf0

SPI: SSPPCELLID2 寄存器

偏移: 0xff8

说明

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 513
SSPPCELLID2 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID2 : 这些位读取值为 0x05	只读	0x05

SPI: SSPPCELLID3 寄存器

偏移: 0ffc

描述

PrimeCell标识寄存器, SSPPCellID0-3, 见第3-16页

表 514。
SSPPCELLID3 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	SSPPCELLID3 : 这些位读回值为 0xB1	只读	0xb1

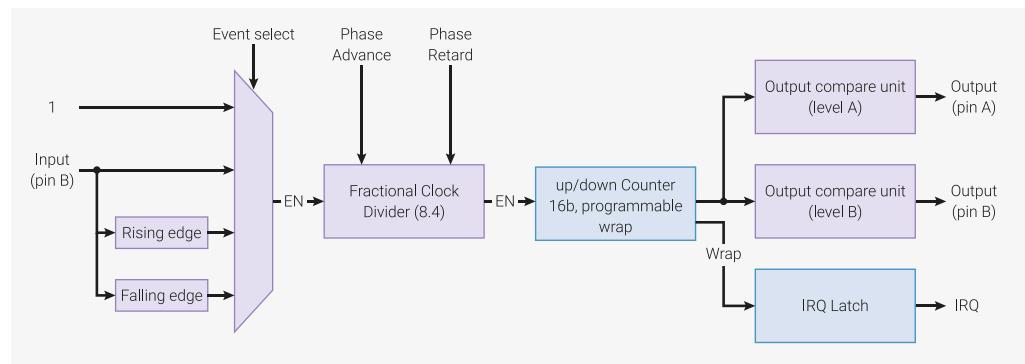
4.5. PWM

4.5.1. 概述

脉冲宽度调制（PWM）是一种通过数字信号提供平滑变化平均电压的方案。该方案通过在固定间隔内施加宽度受控的正脉冲实现。高电平持续时间占总周期的比例称为占空比。该方法可用于模拟输出的近似，或控制开关模式电源电子设备。

RP2040 的 PWM 模块包含8个功能相同的切片。每个切片可驱动两个 PWM 输出信号，或测量输入信号的频率或占空比，故总共可控制多达16个 PWM 输出。所有30个 GPIO 引脚均可由 PWM 模块驱动。

Figure 103. A single PWM slice. A 16-bit counter counts from 0 up to some programmed value, and then wraps to zero, or counts back down again, depending on PWM mode. The A and B outputs transition high and low based on the current count value and the preprogrammed A and B thresholds. The counter advances based on a number of events: it may be free-running, or gated by level or edge of an input signal on the B pin. A fractional divider slows the overall count rate for finer control of output frequency.



Each PWM slice is equipped with the following:

- 16-bit counter
- 8.4 fractional clock divider
- Two independent output channels, duty cycle from 0% to 100% **inclusive**
- Dual slope and trailing edge modulation
- Edge-sensitive input mode for frequency measurement
- Level-sensitive input mode for duty cycle measurement
- Configurable counter wrap value
 - Wrap and level registers are double buffered and can be changed race-free while PWM is running
- Interrupt request and DMA request on counter wrap
- Phase can be precisely advanced or retarded while running (increments of one count)

Slices can be enabled or disabled simultaneously via a single, global control register. The slices then run in perfect lockstep, so that more complex power circuitry can be switched by the outputs of multiple slices.

4.5.2. Programmer's Model

All 30 GPIO pins on RP2040 can be used for PWM:

Table 515. Mapping of PWM channels to GPIO pins on RP2040. This is also shown in the main GPIO function table, Table 279

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

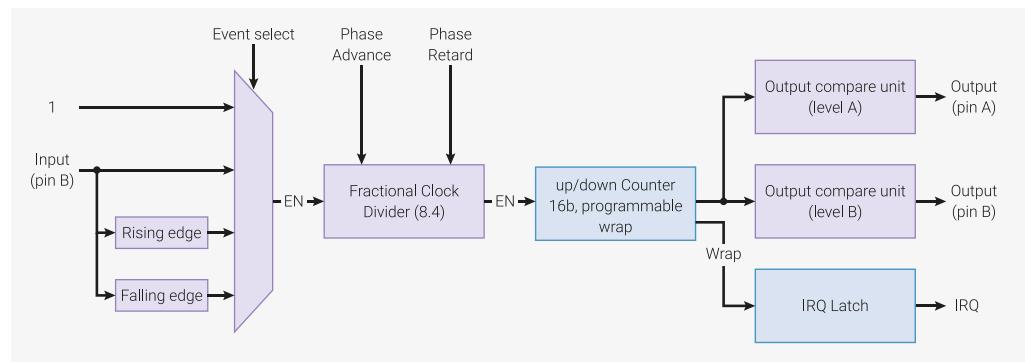
- The 16 PWM channels (8 2-channel slices) appear on GPIO0 to GPIO15, in the order PWM0 A, PWM0 B, PWM1 A...
- This repeats for GPIO16 to GPIO29. GPIO16 is PWM0 A, GPIO17 is PWM0 B, so on up to PWM6 B on GPIO29
- The same PWM output can be selected on two GPIO pins; the same signal will appear on each GPIO.
- If a PWM B pin is used as an input, and is selected on multiple GPIO pins, then the PWM slice will see the logical OR of those two GPIO inputs

4.5.2.1. Pulse Width Modulation

The PWM hardware functions by continuously comparing the input value to a free-running counter. This produces a toggling output where the amount of time spent at the high output level is proportional to the input value. The fraction of time spent at the high signal level is known as the duty cycle of the signal.

图103。一个单个 PWM 切片。一个16位计数器从0计数至设定值，然后根据 PWM 模式环绕回零，或反向计数。A 和 B 输出根据当前计数值及预设的 A 和 B 阀值在高低电平之间切换。计数器基于多个事件递增：它

可以自由运行，或通过 B 引脚输入信号的电平或边沿进行门控。分数时钟除频器降低整体计数速率，以实现更精细的输出频率控制。



每个 PWM 单元配备以下功能：

- 16 位计数器
- 8.4 分数时钟除频器
- 两个独立输出通道，占空比范围为 0% 至 100% 包含端点
- 双斜率及尾部边沿调制
- 用于频率测量的边沿敏感输入模式
- 用于占空比测量的电平敏感输入模式
- 可配置的计数器循环值
 - 循环值和电平寄存器采用双缓冲设计，可在 PWM 运行时无竞争地更新
- 计数器循环时产生中断请求和 DMA 请求
- 相位可在运行时精确提前或延迟（以单计数步进）

可通过单一全局控制寄存器同时启用或禁用所有单元。切片随后以完美同步的方式运行，以便通过多个切片的输出切换更复杂的电源电路。

4.5.2. 程序员模型

RP2040的所有30个GPIO引脚均可用于PWM：

表515。RP2040上PWM通道与GPIO引脚的映射。
该信息亦显示于主GPIO功能表（表279）中。

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM通道	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM通道	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

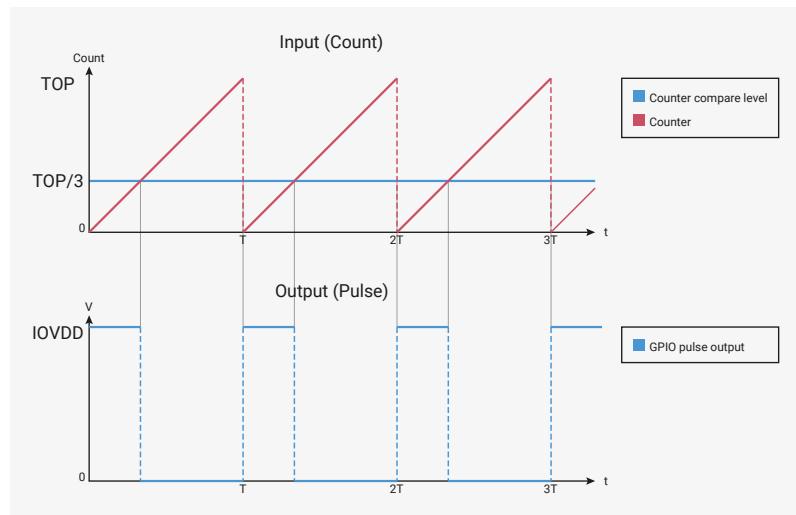
- 这16个PWM通道（8个双通道切片）分布在GPIO0至GPIO15，引脚顺序为PWM0 A、PWM0 B、PWM1 A...
- 此规律重复应用于GPIO16至GPIO29。GPIO16为PWM0 A，GPIO17为PWM0 B，依此类推，至GPIO29的PWM6 B
- 同一PWM信号可被选择输出至两个GPIO引脚；相同信号将同时出现在各GPIO引脚上。
- 若PWM B引脚用作输入且被多个GPIO引脚选中，则PWM切片会接收这些GPIO输入的逻辑或信号

4.5.2.1. 脉冲宽度调制

PWM硬件通过持续比较输入值与自由运行计数器实现其功能。该过程生成切换输出，其中高电平输出时间的长短与输入值成比例。高电平信号维持的时间比例即为信号的占空比。

The counting period is controlled by the **TOP** register, with a maximum possible period of 65536 cycles, as the counter and **TOP** are 16 bits in size. The input values are configured via the **CC** register.

Figure 104. The counter repeatedly counts from 0 to **TOP**, forming a sawtooth shape. The counter is continuously compared with some input value. When the input value is higher than the counter, the output is driven high. Otherwise, the output is low. The output period T is defined by the **TOP** value of the counter, and how fast the counter is configured to count. The average output voltage, as a fraction of the IO power supply, is the input value divided by the counter period ($\text{TOP} + 1$)



This example shows the counting period and the A and B counter compare levels being configured on one of RP2040's PWM slices.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pwm/hello_pwm/hello_pwm.c Lines 14 - 29

```

14 // Tell GPIO 0 and 1 they are allocated to the PWM
15 gpio_set_function(0, GPIO_FUNC_PWM);
16 gpio_set_function(1, GPIO_FUNC_PWM);
17
18 // Find out which PWM slice is connected to GPIO 0 (it's slice 0)
19 uint slice_num = pwm_gpio_to_slice_num(0);
20
21 // Set period of 4 cycles (0 to 3 inclusive)
22 pwm_set_wrap(slice_num, 3);
23 // Set channel A output high for one cycle before dropping
24 pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
25 // Set initial B output high for three cycles before dropping
26 pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
27 // Set the PWM running
28 pwm_set_enabled(slice_num, true);

```

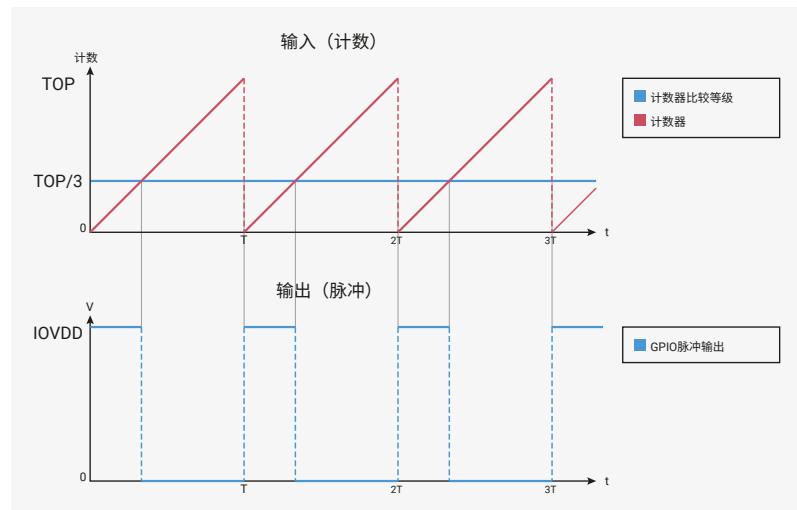
Figure 105 shows how the PWM hardware operates once it has been configured in this way.

计数周期由 **TOP** 寄存器控制，最大周期为 65536 个计数周期，因计数器和 **TOP** 均为 16 位。输入数值通过 **CC** 寄存器进行配置。

图104。计数器反复从0计数至TOP，形成锯齿波形。计数器数值会持续与某一输入值比较。当输入值大于计数器值时，输出为高电平。

否则，输出为低电平。输出周期T由计数器的TOP值及计数速度决定。

作为I/O电源电压的比例，平均输出电压为输入值除以计数周期($TOP + 1$)



本示例展示了如何在RP2040的某个PWM切片上配置计数周期及A、B计数比较电平。

Pico示例：https://github.com/raspberrypi/pico-examples/blob/master/pwm/hello_pwm/hello_pwm.c 第14至29行

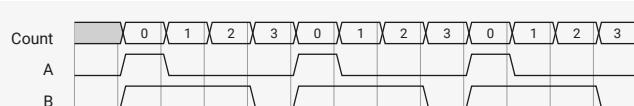
```

14 // 告诉GPIO 0和1它们被分配为PWM功能
15 gpio_set_function(0, GPIO_FUNC_PWM);
16 gpio_set_function(1, GPIO_FUNC_PWM);
17
18 // 查询连接到GPIO 0的PWM切片（即切片0）
19 uint slice_num = pwm_gpio_to_slice_num(0);
20
21 // 将周期设置为4个计数单位（包括0至3）
22 pwm_set_wrap(slice_num, 3);
23 // 设置通道A先保持高电平一个周期然后下降
24 pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
25 // 在下降前将B输出保持高电平三个周期
26 pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
27 // 启动PWM
28 pwm_set_enabled(slice_num, true);

```

图105展示了PWM硬件在此配置方式下的运行情况。

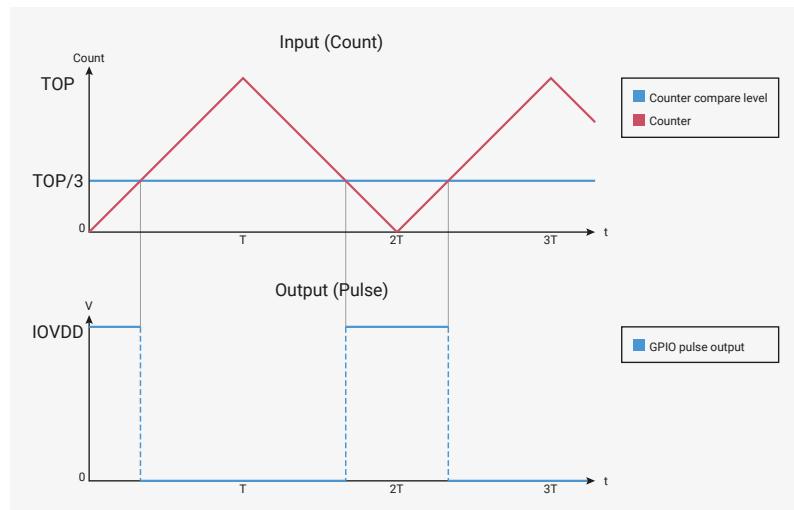
Figure 105. The slice counts repeatedly from 0 to 3, which is configured as the **TOP** value. The output waves therefore have a period of 4. Output A is high for 1 cycle in 4, so the average output voltage is 1/4 of the IO supply voltage. Output B is high for 3 cycles in every 4. Note the rising edges of A and B are always aligned.



The default behaviour of a PWM slice is to count upward until the value of the **TOP** register is reached, and then immediately wrap to 0. PWM slices also offer a phase-correct mode, enabled by setting **CSR_PH_CORRECT** to 1, where the counter starts to count downward after reaching **TOP**, until it reaches 0 again.

It is called phase-correct mode because the pulse is always centred on the same point, no matter the duty cycle. In other words, its phase is not a function of duty cycle. The output frequency is halved when phase-correct mode is enabled.

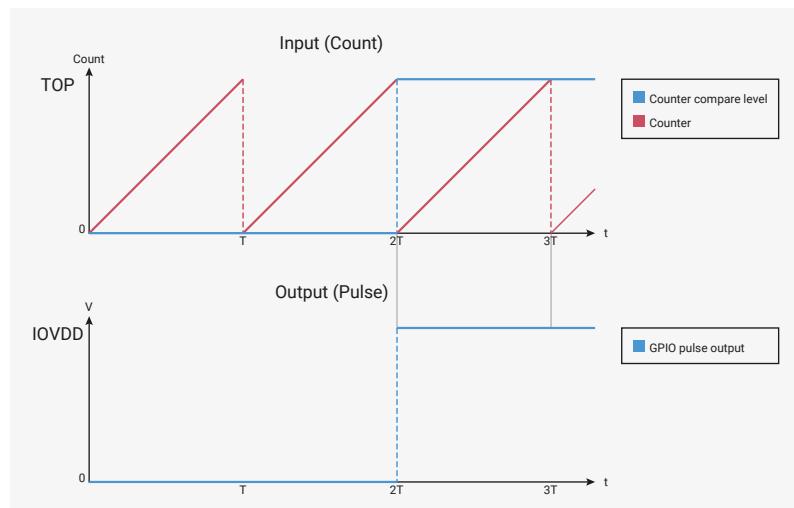
Figure 106. In phase-correct mode, the counter counts back down from **TOP** to 0 once it reaches **TOP**.



4.5.2.2. 0% and 100% Duty Cycle

The RP2040 PWM can produce toggle-free 0% and 100% duty cycle output.

Figure 107. Glitch-free 0% duty cycle output for **CC** = 0, and glitch-free 100% duty cycle output for **CC** = **TOP** + 1



A **CC** value of 0 will produce a 0% output, i.e. the output signal is always low. A **CC** value of **TOP** + 1 (i.e. equal to the period, in non-phase-correct mode) will produce a 100% output. For example, if **TOP** is programmed to 254, the counter will have a period of 255 cycles, and **CC** values in the range of 0 to 255 inclusive will produce duty cycles in the range 0% to 100% inclusive.

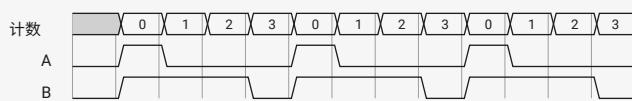
Glitch-free output at 0% and 100% is important e.g. to avoid switching losses when a MOSFET is controlled at its minimum and maximum current levels.

图105。该切片计数从0计数至3，3为配置的TOP值。因此输出波形的周期为4。

输出A在4个周期中高电平持续1个周期，故平均输出电压为IO电源电压的1/4。

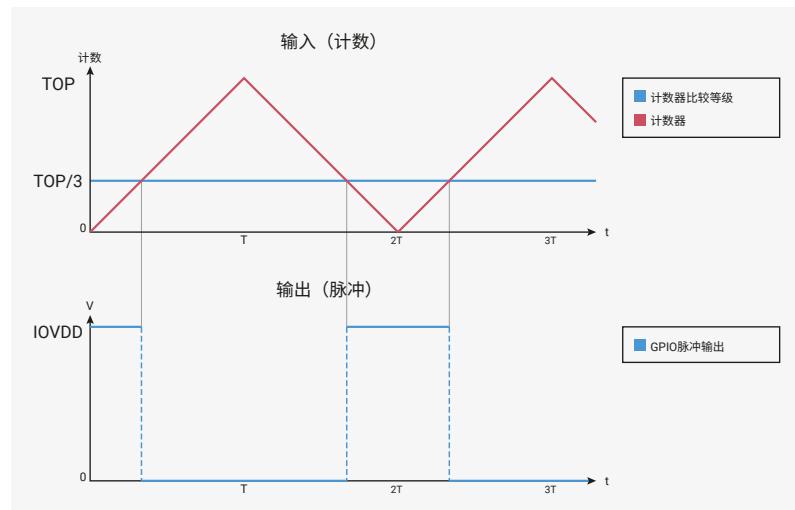
输出B在每4个周期中高电平持续3个周期。注意A和B的上升沿始终保持对齐。

图106。在相位校正模式下，计数器在达到TOP后开始从TOP向下计数至0。



PWM切片的默认行为是向上计数，直至达到 TOP寄存器的设定值，然后立即回绕至0。PWM片段还提供相位校正模式，通过将CSR_PH_CORRECT设置为1启用，该模式下计数器在达到TOP后开始向下计数，直至再次达到0。

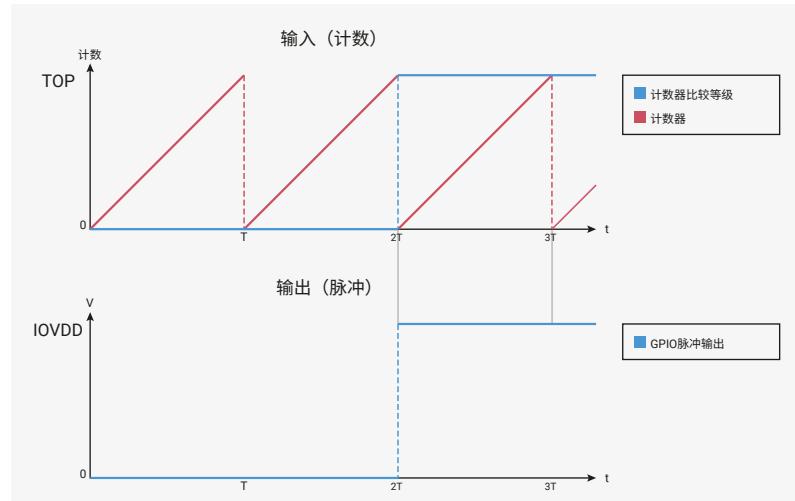
之所以称为相位校正模式，是因为脉冲始终以相同点为中心，无论占空比如何变化。换言之，其相位不依赖于占空比。启用相位校正模式时，输出频率减半。



4.5.2.2 0%和100%占空比

RP2040 PWM能够产生无跳变的0%和100%占空比输出。

图107。当CC = 0时，实现无毛刺的0%占空比输出；当CC = TOP + 1时，实现无毛刺的100%占空比输出。
1



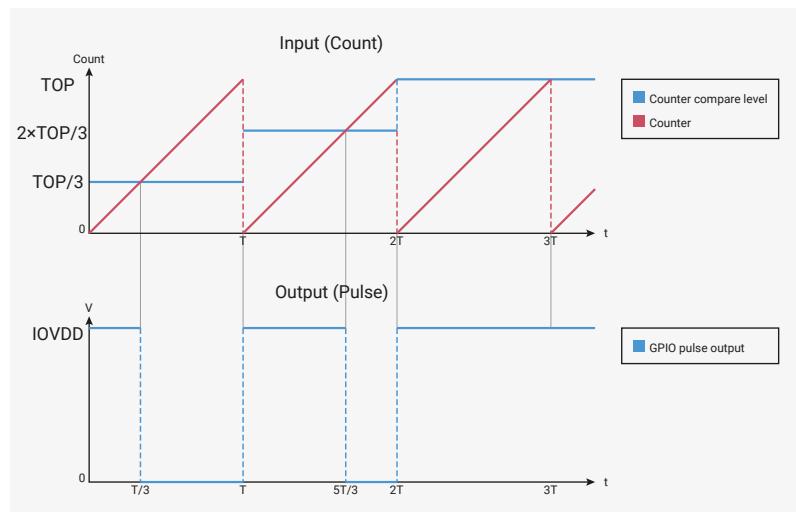
当CC值为0时，将产生0%输出，即输出信号始终为低电平；当CC值为TOP+1（即在非相位校正模式下等于周期）时，将产生100%输出。例如，若TOP设定为254，则计数器周期为255个时钟周期，且CC值在0至255之间（含）将对应0%至100%（含）的占空比。

实现0%和100%的无毛刺输出至关重要，例如避免在MOSFET以其最小和最大电流控制时产生开关损耗。

4.5.2.3. Double Buffering

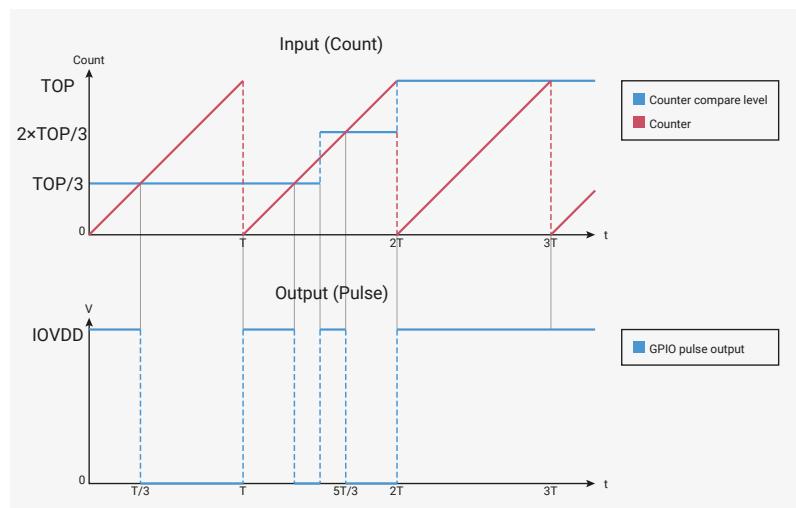
Figure 108 shows how a change in input value will produce a change in output duty cycle. This can be used to approximate some analog waveform such as a sine wave.

Figure 108. The input value varies with each counter period: first $\text{TOP} / 3$, then $2 \times \text{TOP} / 3$, and finally $\text{TOP} + 1$ for 100% duty cycle. Each increase in the input value causes a corresponding increase in the output duty cycle.



In Figure 108, the input value only changes at the instant where the counter wraps through 0. Figure 109 shows what happens if the input value is allowed to change at any other time: an unwanted glitch is produced at the output.

Figure 109. The input value changes whilst the counter is mid-ramp. This produces additional toggling at the output.



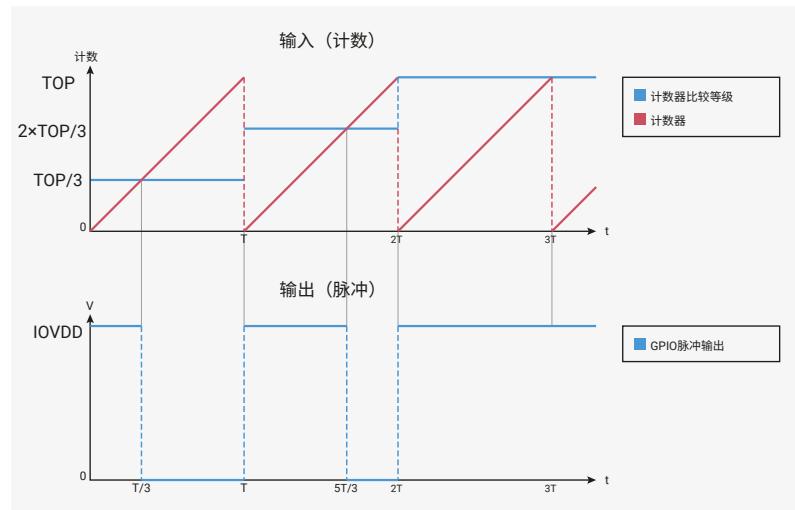
The behaviour becomes even more perplexing if the `TOP` register is also modified. It would be difficult for software to write to `CC` or `TOP` with the correct timing. To solve this, each slice has two copies of the `CC` and `TOP` registers: one copy which software can modify, and another, internal copy which is updated from the first register at the instant the counter wraps. Software can modify its copy of the register at will, but the changes are not captured by the PWM output until the next wrap.

Figure 110 shows the sequence of events where a software interrupt handler changes the value of `CC_A` each time the counter wraps.

4.5.2.3. 双缓冲

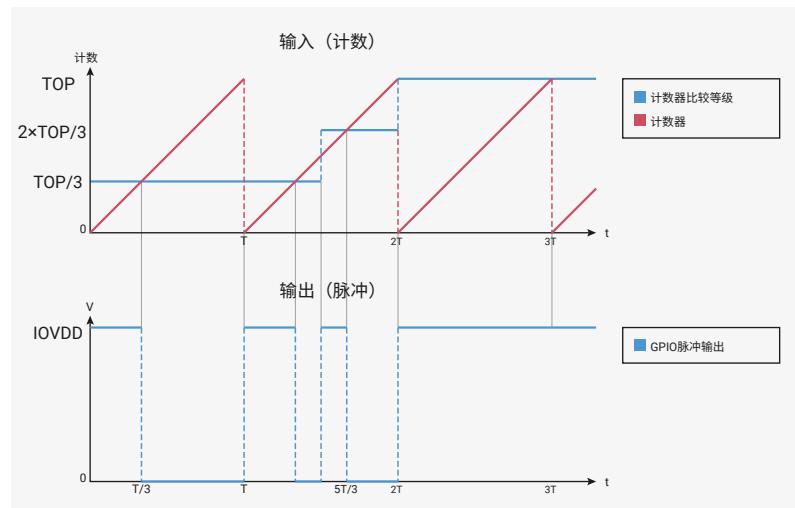
图108显示了输入值的变化如何导致输出占空比的变化。此方法可用于近似某些模拟波形，例如正弦波。

图108。输入值随每个计数周期变化：依次为 $TOP / 3$ ， $2 \times TOP / 3$ ，最后为 $TOP + 1$ ，对应100%的占空比。每次输入值的增加均会引起输出占空比的相应增加。



在图108中，输入值仅在计数器计数回零瞬间发生变化。图109展示了若允许输入值在其他任意时刻改变，输出将产生不期望的毛刺。

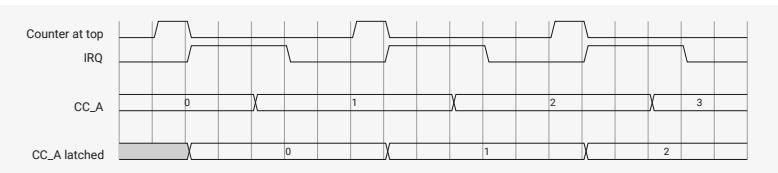
图109。输入值在计数器计数中途时发生变化。这将导致输出出现额外的切换。



如果同时修改 TOP 寄存器，行为将更加复杂难以理解。软件难以以正确时机对 CC 或 TOP 寄存器进行写操作。为解决此问题，每个切片设有两份 CC 和 TOP 寄存器副本：一份由软件修改，另一份为内部副本，在计数器计数回零瞬间从前者更新。软件可随意更改其寄存器副本，但更改仅在下一次计数器回零时才被 PWM 输出捕获。

图110展示了软件中断处理程序在每次计数器回零时更改 CC_A 数值的事件序列。

Figure 110. Each counter wrap causes the interrupt request signal to assert. The processor enters its interrupt handler, writes to its copy of the CC register, and clears the interrupt. When the counter wraps again, the latched version of the CC register is instantaneously updated with the most recent value written by software, and this value controls the duty cycle for the next period. The IRQ is reasserted so that software can write another fresh value to its copy of the CC register.



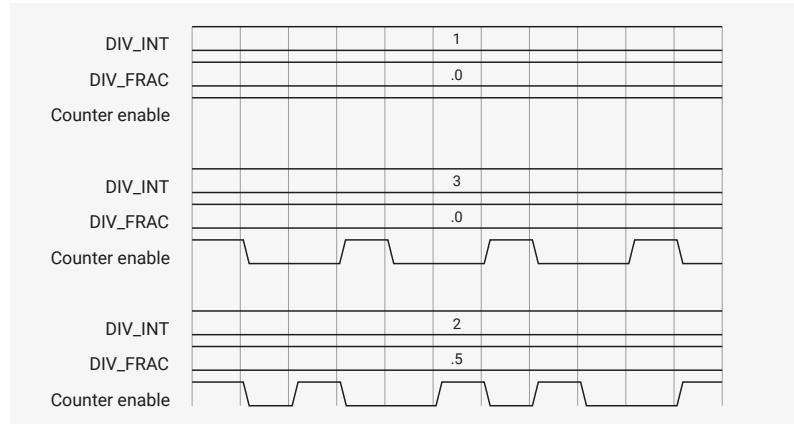
There is no limitation on what values can be written to **CC** or **TOP**, or when they are written. In normal PWM mode (**CSR_PH_CORRECT** is 0) the latched copies are updated when the counter wraps to 0, which occurs once every **TOP + 1** cycles. In phase-correct mode (**CSR_PH_CORRECT** is 1), the latched copies are updated on the 0 to 0 count transition, i.e. the point where the counter stops counting downward and begins to count upward again.

4.5.2.4. Clock Divider

Each slice has a fractional clock divider, configured by the **DIV** register. This is an 8 integer bit, 4 fractional bit clock divider, which allows the count rate to be slowed by up to a factor of 256. The clock divider allows much lower output frequencies to be achieved – approximately 7.5Hz from a 125MHz system clock. Lower frequencies than this will require a system timer interrupt (Section 4.6)

It does this by generating an enable signal which gates the operation of the counter.

Figure 111. The clock divider generates an enable signal. The counter only counts on cycles where this signal is high. A clock divisor of 1 causes the enable to be asserted on every cycle, so the counter counts by one on every system clock cycle. Higher divisors cause the count enable to be asserted less frequently. Fractional division achieves an average fractional counting rate by spacing some enable pulses further apart than others.



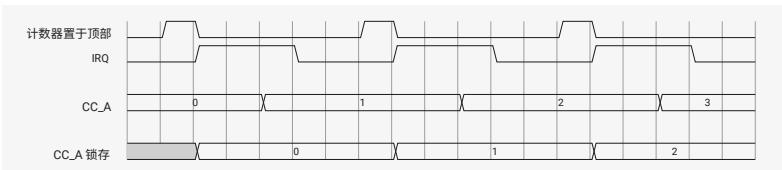
The fractional divider is a first-order delta-sigma type.

The clock divider also allows the effective count range to be extended, when using level-sensitive or edge-sensitive modes to take duty cycle or frequency measurements.

4.5.2.5. Level-sensitive and Edge-sensitive Triggering

图110。每当计数器回绕时，中断请求信号即被置位。处理器进入中断处理程序，向其CC寄存器的副本写入数据，并清除中断。

当计数器再次回绕时，CC寄存器的锁存版本会即时更新为软件最近写入的值，该值将控制下一周期的占空比。中断请求信号再次被置位，以便软件向其CC寄存器副本写入新的值。



对CC或TOP的写入数值及写入时间不受任何限制。在正常PWM模式下（CSR_PH_CORRECT为0），锁存副本在计数器回绕至0时更新，该过程每TOP+1个周期发生一次。在相位校正模式下（CSR_PH_CORRECT为1），锁存副本在0到0的计数转换点更新，即计数器停止向下计数并开始向上计数的时刻。

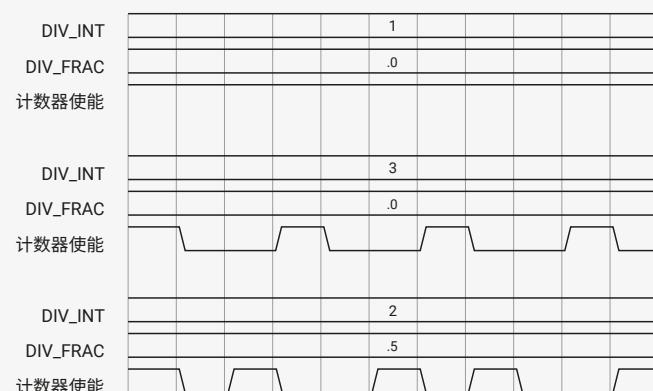
4.5.2.4. 时钟分频器

每个片段具有一个分数时钟分频器，由DIV寄存器配置。这是一个8位整数位和4位小数位的时钟分频器，允许计数速率最多降低256倍。时钟分频器可实现更低的输出频率——从125MHz系统时钟约降至7.5Hz。低于此频率将需要系统定时器中断（第4.6节）。

其工作原理是生成一个使能信号，该信号门控计数器的运作。

图111。时钟分频器生成使能信号。计数器仅在该信号为高电平时计数。分频系数为1时，使能在每个周期断言，计数器在每个系统时钟周期递增1。较高的分频系数会降低计数使能的断言频率。

分数分频通过延长部分使能脉冲间隔，实现平均分数计数速率。

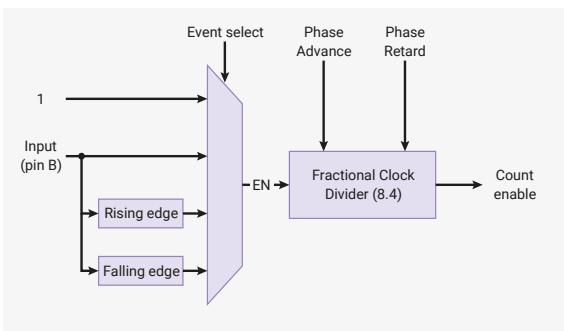


分数除法器属于一阶delta-sigma型。

时钟除法器还允许在使用电平敏感或边沿敏感模式进行占空比或频率测量时，有效计数范围得以扩展。

4.5.2.5. 电平敏感与边沿敏感触发

Figure 112. PWM slice event selection. The counter advances when its enable input is high, and this enable is generated in two sequential stages. First, any one of four event types (always on, pin B high, pin B rise, pin B fall) can generate enable pulses for the fractional clock divider. The divider can reduce the rate of the enable pulses, before passing them on to the counter.



By default, each slice's counter is free-running, and will count continuously whenever the slice is enabled. There are three other options available:

- Count continuously when a high level is detected on the B pin
- Count once with each rising edge detected on the B pin
- Count once with each falling edge detected on the B pin

These modes are selected by the `DIVMODE` field in each slice's `CSR`. In free-running mode, the A and B pins are both outputs. In any other mode, the B pin becomes an input, and controls the operation of the counter. `CC_B` is ignored when not in free-running mode.

By allowing the slice to run for a fixed amount of time in level-sensitive or edge-sensitive mode, it's possible to measure the duty cycle or frequency of an input signal. Due to the type of edge-detect circuit used, the low period and high period of the measured signal must both be strictly greater than the system clock period when taking frequency measurements.

The clock divider is still operational in level-sensitive and edge-sensitive mode. At maximum division (writing 0 to `DIV_INT`), the counter will only advance once per 256 high input cycles in level-sensitive modes, or once per 256 edges in edge-sensitive mode. This allows longer-running measurements to be taken, although the resolution is still just 16 bits.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pwm/measure_duty_cycle/measure_duty_cycle.c Lines 19 - 37

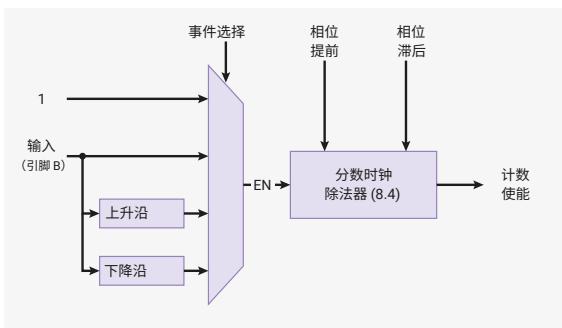
```

19 float measure_duty_cycle(uint gpio) {
20     // Only the PWM B pins can be used as inputs.
21     assert(pwm_gpio_to_channel(gpio) == PWM_CHAN_B);
22     uint slice_num = pwm_gpio_to_slice_num(gpio);
23
24     // Count once for every 100 cycles the PWM B input is high
25     pwm_config cfg = pwm_get_default_config();
26     pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_HIGH);
27     pwm_config_set_clkdiv(&cfg, 100);
28     pwm_init(slice_num, &cfg, false);
29     gpio_set_function(gpio, GPIO_FUNC_PWM);
30
31     pwm_set_enabled(slice_num, true);
32     sleep_ms(10);
33     pwm_set_enabled(slice_num, false);
34     float counting_rate = clock_get_hz(clk_sys) / 100;
35     float max_possible_count = counting_rate * 0.01;
36     return pwm_get_counter(slice_num) / max_possible_count;
37 }
  
```

4.5.2.6. Configuring PWM Period

When free-running, the period of a PWM slice's output (measured in system clock cycles) is controlled by three parameters:

图 112. PWM 分片事件选择。当其使能输入为高电平时，计数器递增，该使能信号由两个连续阶段生成。首先，四种事件类型之一（始终开启、引脚 B 高电平、引脚 B 上升沿、引脚 B 下降沿）可为分数时钟除法器生成使能脉冲。除法器可降低使能脉冲的频率，再将脉冲传递给计数器。



默认情况下，每个分片的计数器均为自由运行状态，只要分片被使能，计数器将连续计数。还有另外三种可选项：

- 当B脚检测到高电平时连续计数
- 每当检测到B脚的上升沿时计数一次
- 每当检测到B脚的下降沿时计数一次

这些模式由各切片CSR寄存器中的DIVMODE字段选择。在自由运行模式下，A脚和B脚均为输出。在其他任何模式下，B脚变为输入，并控制计数器的运行。非自由运行模式下，CC_B字段被忽略。

通过允许切片在电平敏感或边缘敏感模式下运行特定时间，可测量输入信号的占空比或频率。由于所用边缘检测电路的特性，频率测量时，信号的高电平周期和低电平周期均必须严格大于系统时钟周期。

时钟分频器在电平敏感和边缘敏感模式下依然有效。在最大分频（向DIV_INT写入0）时，计数器在电平敏感模式下每256个高电平输入周期仅递增一次，或在边缘敏感模式下每256个边沿递增一次。该设置允许执行更长时间的测量，尽管分辨率仍为16位。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/pwm/measure_duty_cycle/measure_duty_cycle.c 第19至37行

```

19 float measure_duty_cycle(uint gpio) {
20     // 仅PWM通道B的引脚可用于输入。
21     assert(pwm_gpio_to_channel(gpio) == PWM_CHAN_B);
22     uint slice_num = pwm_gpio_to_slice_num(gpio);
23
24     // 每当PWM通道B输入保持高电平100个周期时计数一次
25     pwm_config cfg = pwm_get_default_config();
26     pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_HIGH);
27     pwm_config_set_clkdiv(&cfg, 100);
28     pwm_init(slice_num, &cfg, false);
29     gpio_set_function(gpio, GPIO_FUNC_PWM);
30
31     pwm_set_enabled(slice_num, true);
32     sleep_ms(10);
33     pwm_set_enabled(slice_num, false);
34     float counting_rate = clock_get_hz(clk_sys) / 100;
35     float max_possible_count = counting_rate * 0.01;
36     return pwm_get_counter(slice_num) / max_possible_count;
37 }
```

4.5.2.6. 配置 PWM 周期

在自由运行模式下，PWM 片段输出的周期（以系统时钟周期计）由以下三个参数控制：

- The **TOP** register
- Whether phase-correct mode is enabled (**CSR_PH_CORRECT**)
- The **DIV** register

The slice counts from 0 to **TOP**, and then either wraps, or begins counting backward, depending on the setting of **CSR_PH_CORRECT**. The rate of counting is slowed by the clock divider, with a maximum speed of one count per cycle, and a minimum speed of one count per $\frac{255}{16}$ ¹⁵ cycles. The period in clock cycles can be calculated as:

$$\text{period} = (\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)$$

The output frequency can then be determined based on the system clock frequency:

$$f_{\text{PWM}} = \frac{f_{\text{sys}}}{\text{period}} = \frac{f_{\text{sys}}}{(\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)}$$

4.5.2.7. Interrupt Request (IRQ) and DMA Data Request (DREQ)

The PWM block has a single IRQ output. The interrupt status registers **INTR**, **INTS** and **INTE** allow software to control which slices will assert this IRQ output, to check which slices are the cause of the IRQ's assertion, and to clear and acknowledge the interrupt.

A slice generates an interrupt request each time its counter wraps (or, if **CSR_PH_CORRECT** is enabled, each time the counter returns to 0). This sets the flag corresponding to this slice in the raw interrupt status register, **INTR**. If this slice's interrupt is enabled in **INTE**, then this flag will cause the PWM block's IRQ to be asserted, and the flag will also appear in the masked interrupt status register **INTS**.

Flags are cleared by writing a mask back to **INTR**. This is demonstrated in the "LED fade" SDK example.

This scheme allows multiple slices to generate interrupts concurrently, and a system interrupt handler to determine which slices caused the most recent interruption, and handle appropriately. Normally this would mean reloading those slices' **TOP** or **CC** registers, but the PWM block can also be used as a source of regular interrupt requests for non-PWM-related purposes.

The same pulse which sets the interrupt flag in **INTR** is also available as a one-cycle data request to the RP2040 system DMA. For each cycle the DMA sees a DREQ asserted, it will make one data transfer to its programmed location, in as timely a manner as possible. In combination with the double-buffered behaviour of **CC** and **TOP**, this allows the DMA to efficiently stream data to a PWM slice at a rate of one transfer per counter period. Alternatively, a PWM slice could serve as a pacing timer for DMA transfers to some other memory-mapped hardware.

4.5.2.8. On-the-fly Phase Adjustment

For some applications it is necessary to control the phase relationship between two PWM outputs on different slices.

The global enable register **EN** contains an alias of the **CSR_EN** flag for each slice, and allows multiple slices to be started and stopped simultaneously. If two slices with the same output frequency are started at the same time, they will run in perfect lockstep, and have a fixed phase relationship, determined by the initial counter values.

The **CSR_PH_ADV** and **CSR_PH_RET** fields will advance or retard a slice's output phase by one count, whilst it is running. They do so by inserting or deleting pulses from the clock enable (the output of the clock divider), as shown in [Figure 113](#).

- **TOP** 寄存器
- 相位校正模式是否启用 (**CSR_PH_CORRECT**)
- **DIV** 寄存器

计数器从 0 计数到 **TOP**，然后根据 **CSR_PH_CORRECT** 的设置，要回绕，要开始反向计数。计数速率由时钟分频器降低，最大速度为每周期一次计数，

最小速度为每 $\frac{15}{255}$ 周期一次计数。时钟周期内的周期可计算为：

$$\text{period} = (\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)$$

然后可根据系统时钟频率确定输出频率：

$$f_{\text{PWM}} = \frac{f_{\text{sys}}}{\text{period}} = \frac{f_{\text{sys}}}{(\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)}$$

4.5.2.7. 中断请求 (IRQ) 与DMA数据请求 (DREQ)

PWM模块有一个单一的IRQ输出。中断状态寄存器 **INTR**、**INTS**和**INTE**允许软件控制哪些切片会触发该IRQ输出，检查引起IRQ的切片，并清除与确认中断。

每当切片计数器回绕（或启用**CSR_PH_CORRECT**时计数器返回至0）时，该切片将生成一个中断请求。这将在原始中断状态寄存器 **INTR**中设置对应该切片的标志。若该切片的中断在 **INTE**中被启用，该标志将触发PWM模块的IRQ，并且该标志亦会出现在屏蔽中断状态寄存器 **INTS**中。

通过向 **INTR**写入掩码即可清除标志。此内容已在“LED fade” SDK示例中演示。

该方案允许多个切片同时产生中断，系统中断处理程序能够确定最近期的中断由哪些切片引起，并进行相应处理。通常这意味着需要重新加载这些切片的 **TOP**或**CC**寄存器，但PWM模块也可作为非PWM相关用途的定期中断请求源。

在 **INTR**中设置中断标志的脉冲，同时也可作为RP2040系统DMA的单周期数据请求信号。DMA每检测到DREQ有效一个周期，即会尽可能及时地向其预设位置执行一次数据传输。结合 **CC**与 **TOP**的双缓冲特性，DMA能够以每个计数周期一次传输的速率，高效地向PWM切片进行数据流传输。或者，PWM切片也可用作DMA向其他内存映射硬件传输的节拍定时器。

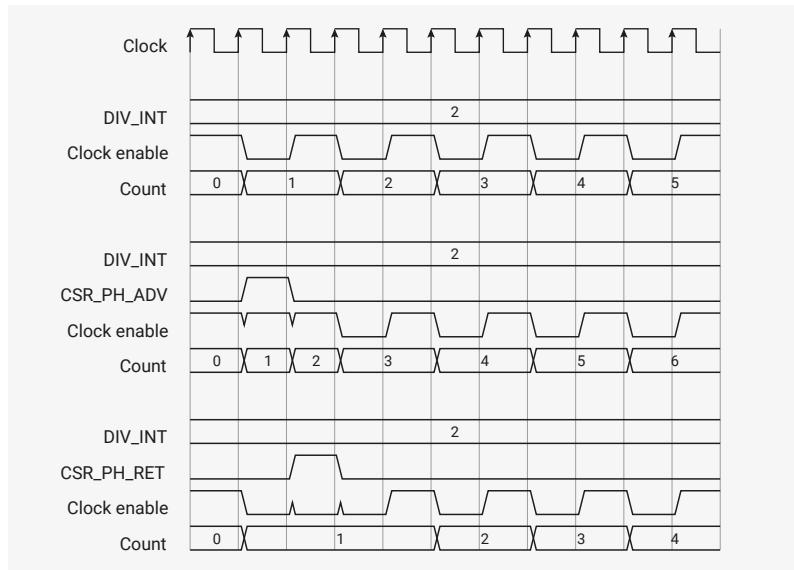
4.5.2.8. 即时相位调整

对于某些应用，需控制不同切片上两个PWM输出的相位关系。

全局使能寄存器 **EN**包含每个切片的 **CSR_EN**标志别名，允许多个切片同时启动和停止。若两个具有相同输出频率的切片同时启动，将实现完美锁步运行，并保持由初始计数器值决定的固定相位关系。

CSR_PH_ADV和**CSR_PH_RET**字段能在切片运行时使其输出相位提前或延迟一个计数。其方法是在时钟使能（即时钟分频器输出）中插入或删除脉冲，如图113所示。

Figure 113. The clock enable signal, output by the clock divider, controls the rate of counting. Phase advance forces the clock enable high on cycles where it is low, causing the counter to jump forward by one count. Phase retard forces the clock enable low when it would be high, holding the counter back by one count.



The counter can not count faster than once per cycle, so **PH_ADV** requires **DIV_INT** > 1 or **DIV_FRAC** > 0. Likewise, the counter will not start to count backward if **PH_RET** is asserted when the clock enable is permanently low.

To advance or retard the phase by one count, software writes 1 to **PH_ADV** or **PH_RET**. Once an enable pulse has been inserted or deleted, the **PH_ADV** or **PH_RET** register bit will return to 0, and software can poll the **CSR** until this happens. **PH_ADV** will always insert a pulse into the next available gap, and **PH_RET** will always delete the next available pulse.

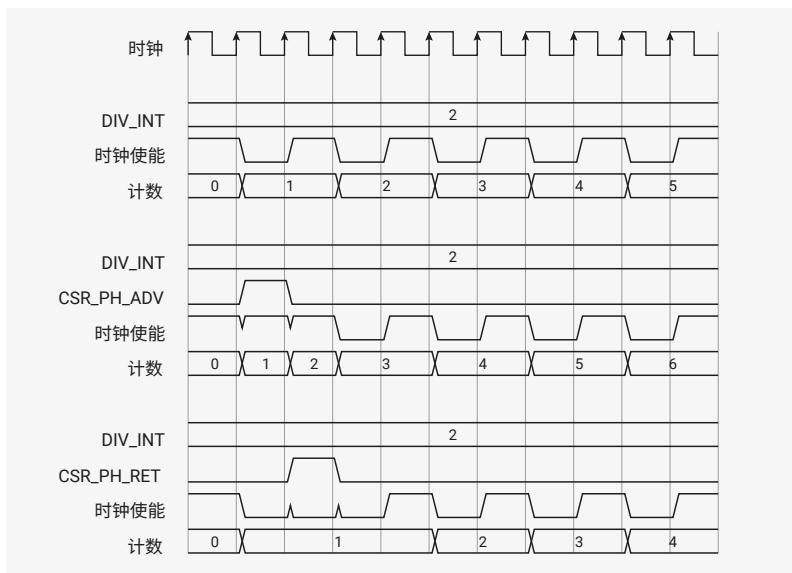
4.5.3. List of Registers

The PWM registers start at a base address of **0x40050000** (defined as **PWM_BASE** in SDK).

Table 516. List of PWM registers

Offset	Name	Info
0x00	CH0_CSR	Control and status register
0x04	CH0_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x08	CH0_CTR	Direct access to the PWM counter
0x0c	CH0_CC	Counter compare values
0x10	CH0_TOP	Counter wrap value
0x14	CH1_CSR	Control and status register
0x18	CH1_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x1c	CH1_CTR	Direct access to the PWM counter
0x20	CH1_CC	Counter compare values
0x24	CH1_TOP	Counter wrap value
0x28	CH2_CSR	Control and status register
0x2c	CH2_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.

图113。时钟使能信号由时钟分频器输出，用以控制计数速率。相位提前通过在时钟使能为低电平时的周期内将其强制置为高电平，使计数器跳跃前进一个计数。相位延迟使时钟使能在应为高电平时变为低电平，从而使计数器回退一个计数。



计数器不能在一个周期内计数超过一次，因此 **PH_ADV** 要求 **DIV_INT > 1** 或 **DIV_FRAC > 0**。同理，若在时钟使能持续为低电平时断言 **PH_RET**，计数器不会开始向后计数。

要使相位前进或延迟一个计数，软件需向 **PH_ADV** 或 **PH_RET** 写入1。一旦插入或删除一个使能脉冲，**PH_ADV** 或 **PH_RET** 寄存器位将恢复为0，软件可轮询 **CSR** 直至此状态。**PH_ADV** 总是在下一个可用空隙插入一个脉冲，而 **PH_RET** 总是删除下一个可用脉冲。

4.5.3. 寄存器列表

PWM寄存器起始地址为 **0x40050000** (SDK中定义为**PWM_BASE**)。

表516。
PWM寄存器列表

偏移量	名称	说明
0x00	CH0_CSR	控制与状态寄存器
0x04	CH0_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x08	CH0_CTR	直接访问 PWM 计数器
0x0c	CH0_CC	计数器比较值
0x10	CH0_TOP	计数器回绕值
0x14	CH1_CSR	控制与状态寄存器
0x18	CH1_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x1c	CH1_CTR	直接访问 PWM 计数器
0x20	CH1_CC	计数器比较值
0x24	CH1_TOP	计数器回绕值
0x28	CH2_CSR	控制与状态寄存器
0x2c	CH2_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。

Offset	Name	Info
0x30	CH2_CTR	Direct access to the PWM counter
0x34	CH2_CC	Counter compare values
0x38	CH2_TOP	Counter wrap value
0x3c	CH3_CSR	Control and status register
0x40	CH3_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x44	CH3_CTR	Direct access to the PWM counter
0x48	CH3_CC	Counter compare values
0x4c	CH3_TOP	Counter wrap value
0x50	CH4_CSR	Control and status register
0x54	CH4_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x58	CH4_CTR	Direct access to the PWM counter
0x5c	CH4_CC	Counter compare values
0x60	CH4_TOP	Counter wrap value
0x64	CH5_CSR	Control and status register
0x68	CH5_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x6c	CH5_CTR	Direct access to the PWM counter
0x70	CH5_CC	Counter compare values
0x74	CH5_TOP	Counter wrap value
0x78	CH6_CSR	Control and status register
0x7c	CH6_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x80	CH6_CTR	Direct access to the PWM counter
0x84	CH6_CC	Counter compare values
0x88	CH6_TOP	Counter wrap value
0x8c	CH7_CSR	Control and status register
0x90	CH7_DIV	INT and FRAC form a fixed-point fractional number. Counting rate is system clock frequency divided by this number. Fractional division uses simple 1st-order sigma-delta.
0x94	CH7_CTR	Direct access to the PWM counter
0x98	CH7_CC	Counter compare values
0x9c	CH7_TOP	Counter wrap value

偏移量	名称	说明
0x30	CH2_CTR	直接访问 PWM 计数器
0x34	CH2_CC	计数器比较值
0x38	CH2_TOP	计数器回绕值
0x3c	CH3_CSR	控制与状态寄存器
0x40	CH3_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x44	CH3_CTR	直接访问 PWM 计数器
0x48	CH3_CC	计数器比较值
0x4c	CH3_TOP	计数器回绕值
0x50	CH4_CSR	控制与状态寄存器
0x54	CH4_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x58	CH4_CTR	直接访问 PWM 计数器
0x5c	CH4_CC	计数器比较值
0x60	CH4_TOP	计数器回绕值
0x64	CH5_CSR	控制与状态寄存器
0x68	CH5_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x6c	CH5_CTR	直接访问 PWM 计数器
0x70	CH5_CC	计数器比较值
0x74	CH5_TOP	计数器回绕值
0x78	CH6_CSR	控制与状态寄存器
0x7c	CH6_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x80	CH6_CTR	直接访问 PWM 计数器
0x84	CH6_CC	计数器比较值
0x88	CH6_TOP	计数器回绕值
0x8c	CH7_CSR	控制与状态寄存器
0x90	CH7_DIV	INT 和 FRAC 组成一个定点小数部分。 计数速率为系统时钟频率除以该数值。 分数除法采用简单的一阶Σ-Δ调制。
0x94	CH7_CTR	直接访问 PWM 计数器
0x98	CH7_CC	计数器比较值
0x9c	CH7_TOP	计数器回绕值

Offset	Name	Info
0xa0	EN	This register aliases the CSR_EN bits for all channels. Writing to this register allows multiple channels to be enabled or disabled simultaneously, so they can run in perfect sync. For each channel, there is only one physical EN register bit, which can be accessed through here or CHx_CSR.
0xa4	INTR	Raw Interrupts
0xa8	INTE	Interrupt Enable
0xac	INTF	Interrupt Force
0xb0	INTS	Interrupt status after masking & forcing

PWM: CH0_CSR, CH1_CSR, ..., CH6_CSR, CH7_CSR Registers

Offsets: 0x00, 0x14, ..., 0x78, 0x8c

Description

Control and status register

Table 517. CH0_CSR, CH1_CSR, ..., CH6_CSR, CH7_CSR Registers

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	PH_ADV : Advance the phase of the counter by 1 count, while it is running. Self-clearing. Write a 1, and poll until low. Counter must be running at less than full speed ($\text{div_int} + \text{div_frac} / 16 > 1$)	SC	0x0
6	PH_RET : Retard the phase of the counter by 1 count, while it is running. Self-clearing. Write a 1, and poll until low. Counter must be running.	SC	0x0
5:4	DIVMODE	RW	0x0
	Enumerated values:		
	0x0 → DIV: Free-running counting at rate dictated by fractional divider		
	0x1 → LEVEL: Fractional divider operation is gated by the PWM B pin.		
	0x2 → RISE: Counter advances with each rising edge of the PWM B pin.		
	0x3 → FALL: Counter advances with each falling edge of the PWM B pin.		
3	B_INV : Invert output B	RW	0x0
2	A_INV : Invert output A	RW	0x0
1	PH_CORRECT : 1: Enable phase-correct modulation. 0: Trailing-edge	RW	0x0
0	EN : Enable the PWM channel.	RW	0x0

PWM: CH0_DIV, CH1_DIV, ..., CH6_DIV, CH7_DIV Registers

Offsets: 0x04, 0x18, ..., 0x7c, 0x90

Description

INT and FRAC form a fixed-point fractional number.

Counting rate is system clock frequency divided by this number.

Fractional division uses simple 1st-order sigma-delta.

偏移量	名称	说明
0xa0	EN	该寄存器为所有通道的 CSR_EN 位设置别名。 写入此寄存器可同时启用或禁用多个通道，实现完美同步运行。 每个通道仅有一个物理 EN 寄存器位， 可通过此处或 CHx_CSR 访问。
0xa4	INTR	原始中断
0xa8	INTE	中断使能
0xac	INTF	中断强制
0xb0	INTS	掩码及强制后的中断状态

PWM: CH0_CSR、CH1_CSR、...、CH6_CSR、CH7_CSR 寄存器

偏移量: 0x00, 0x14, ..., 0x78, 0x8c

描述

控制与状态寄存器

表 517。CH0_CSR、CH1_CSR、...、CH6_CSR、CH7_CSR
R 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	PH_ADV : 在计数器运行时，将计数器相位提前 1 个计数。 自清零。写入1，并轮询直至变为低电平。计数器必须以低于满速 (div_int + div_frac / 16 > 1) 运行。	SC	0x0
6	PH_RET : 在计数器运行时，将其相位延迟1个计数。 自清零。写入1，并轮询直至变为低电平。计数器必须处于运行状态。	SC	0x0
5:4	DIVMODE	读写	0x0
	枚举值：		
	0x0 → DIV：以分数分频器确定的速率自由运行计数。		
	0x1 → LEVEL：分数分频器操作受PWM B引脚门控。		
	0x2 → RISE：计数器随PWM B引脚的每个上升沿递增。		
	0x3 → FALL：计数器随PWM B引脚的每个下降沿递增。		
3	B_INV : 反转输出B。	读写	0x0
2	A_INV : 反转输出A。	读写	0x0
1	PH_CORRECT : 1: 启用相位校正调制；0: 后沿调制。	读写	0x0
0	EN : 启用PWM通道。	读写	0x0

PWM: CH0_DIV、CH1_DIV、.....、CH6_DIV、CH7_DIV 寄存器

偏移量: 0x04、0x18、.....、0x7c、0x90

描述

INT 和 FRAC 组成一个定点小数部分。
计数速率为系统时钟频率除以该数值。
分数除法采用简单的一阶Σ-Δ调制。

*Table 518. CH0_DIV,
CH1_DIV, ..., CH6_DIV,
CH7_DIV Registers*

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:4	INT	RW	0x01
3:0	FRAC	RW	0x0

PWM: CH0_CTR, CH1_CTR, ..., CH6_CTR, CH7_CTR Registers

Offsets: 0x08, 0x1c, ..., 0x80, 0x94

*Table 519. CH0_CTR,
CH1_CTR, ...,
CH6_CTR, CH7_CTR
Registers*

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Direct access to the PWM counter	RW	0x0000

PWM: CH0_CC, CH1_CC, ..., CH6_CC, CH7_CC Registers

Offsets: 0x0c, 0x20, ..., 0x84, 0x98

Description

Counter compare values

*Table 520. CH0_CC,
CH1_CC, ..., CH6_CC,
CH7_CC Registers*

Bits	Description	Type	Reset
31:16	B	RW	0x0000
15:0	A	RW	0x0000

PWM: CH0_TOP, CH1_TOP, ..., CH6_TOP, CH7_TOP Registers

Offsets: 0x10, 0x24, ..., 0x88, 0x9c

*Table 521. CH0_TOP,
CH1_TOP, ...,
CH6_TOP, CH7_TOP
Registers*

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Counter wrap value	RW	0xffff

PWM: EN Register

Offset: 0xa0

Description

This register aliases the CSR_EN bits for all channels.

Writing to this register allows multiple channels to be enabled or disabled simultaneously, so they can run in perfect sync.

For each channel, there is only one physical EN register bit, which can be accessed through here or CHx_CSR.

Table 522. EN Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	CH7	RW	0x0
6	CH6	RW	0x0
5	CH5	RW	0x0
4	CH4	RW	0x0
3	CH3	RW	0x0

表 518。CH0_DIV、CH1_DIV、.....CH6_DIV
、CH7_DIV 寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:4	中断	读写	0x01
3:0	FRAC	读写	0x0

PWM：CH0_CTR、CH1_CTR、.....CH6_CTR、CH7_CTR 寄存器

偏移量：0x08、0x1c、.....0x80、0x94

表 519。CH0_CTR、CH1_CTR、.....CH6_CTR、CH7_CTR 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	直接访问 PWM 计数器	读写	0x0000

PWM：CH0_CC、CH1_CC、.....CH6_CC、CH7_CC 寄存器

偏移量：0x0c、0x20、.....0x84、0x98

描述

计数器比较值

表 520。CH0_CC、CH1_CC、.....CH6_CC
、CH7_CC 寄存器

位	描述	类型	复位值
31:16	B	读写	0x0000
15:0	A	读写	0x0000

PWM：CH0_TOP、CH1_TOP、...、CH6_TOP、CH7_TOP 寄存器

偏移量：0x10、0x24、...、0x88、0x9c

表 521。CH0_TOP、CH1_TOP、...、CH6_TOP、CH7_TOP
P 寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	计数器回绕值	读写	0xffff

PWM：EN 寄存器

偏移：0xa0

描述

该寄存器为所有通道的 CSR_EN 位设置别名。

写入此寄存器可同时启用或禁用多个通道，实现完美同步运行。

每个通道仅有一个物理 EN 寄存器位，
可通过此处或 CHx_CSR 访问。

表 522。EN 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0

Bits	Description	Type	Reset
2	CH2	RW	0x0
1	CH1	RW	0x0
0	CH0	RW	0x0

PWM: INTR Register

Offset: 0xa4

Description

Raw Interrupts

Table 523. INTR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	CH7	WC	0x0
6	CH6	WC	0x0
5	CH5	WC	0x0
4	CH4	WC	0x0
3	CH3	WC	0x0
2	CH2	WC	0x0
1	CH1	WC	0x0
0	CH0	WC	0x0

PWM: INTE Register

Offset: 0xa8

Description

Interrupt Enable

Table 524. INTE Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	CH7	RW	0x0
6	CH6	RW	0x0
5	CH5	RW	0x0
4	CH4	RW	0x0
3	CH3	RW	0x0
2	CH2	RW	0x0
1	CH1	RW	0x0
0	CH0	RW	0x0

PWM: INTF Register

Offset: 0xac

位	描述	类型	复位值
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTR寄存器

偏移: 0xa4

描述

原始中断

表 523. INTR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	WC	0x0
6	CH6	WC	0x0
5	CH5	WC	0x0
4	CH4	WC	0x0
3	CH3	WC	0x0
2	CH2	WC	0x0
1	CH1	WC	0x0
0	CH0	WC	0x0

PWM: INTF 寄存器

偏移: 0xac

描述

中断使能

表 524. INTF
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTE 寄存器

偏移: 0xa8

Description

Interrupt Force

Table 525. INTF Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	CH7	RW	0x0
6	CH6	RW	0x0
5	CH5	RW	0x0
4	CH4	RW	0x0
3	CH3	RW	0x0
2	CH2	RW	0x0
1	CH1	RW	0x0
0	CH0	RW	0x0

PWM: INTS Register

Offset: 0xb0

Description

Interrupt status after masking & forcing

Table 526. INTS Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7	CH7	RO	0x0
6	CH6	RO	0x0
5	CH5	RO	0x0
4	CH4	RO	0x0
3	CH3	RO	0x0
2	CH2	RO	0x0
1	CH1	RO	0x0
0	CH0	RO	0x0

4.6. Timer

4.6.1. Overview

The system timer peripheral on RP2040 provides a global microsecond timebase for the system, and generates interrupts based on this timebase. It supports the following features:

- A single 64-bit counter, incrementing once per microsecond
- This counter can be read from a pair of latching registers, for race-free reads over a 32-bit bus.
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

The timer uses a one microsecond reference that is generated in the Watchdog (see [Section 4.7.2](#)), and derived from

描述

中断强制

表 525. INTF
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	读写	0x0
6	CH6	读写	0x0
5	CH5	读写	0x0
4	CH4	读写	0x0
3	CH3	读写	0x0
2	CH2	读写	0x0
1	CH1	读写	0x0
0	CH0	读写	0x0

PWM: INTS 寄存器

偏移: 0xb0

描述

掩码及强制后的中断状态

表 526. INTS
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7	CH7	只读	0x0
6	CH6	只读	0x0
5	CH5	只读	0x0
4	CH4	只读	0x0
3	CH3	只读	0x0
2	CH2	只读	0x0
1	CH1	只读	0x0
0	CH0	只读	0x0

4.6. 定时器**4.6.1. 概述**

RP2040上的系统定时器外设为系统提供全局微秒时间基准，并基于该时间基准产生中断。其支持以下功能：

- 一个64位单计数器，每微秒递增一次
- 该计数器可通过一对锁存寄存器读取，实现基于32位总线的无竞争读取。
- 四个报警：匹配计数器低32位时触发中断请求（IRQ）。

该定时器采用在看门狗中生成的一微秒参考时钟（详见第4.7.2节）作为基准，并由此派生。

the reference clock (Figure 28), which itself is usually connected directly to the crystal oscillator (Section 2.16).

The 64-bit counter effectively can not overflow (thousands of years at 1MHz), so the system timer is completely monotonic in practice.

4.6.1.1. Other Timer Resources on RP2040

The system timer is intended to provide a global timebase for software. RP2040 has a number of other programmable counter resources which can provide regular interrupts, or trigger DMA transfers.

- The PWM (Section 4.5) contains 8x 16-bit programmable counters, which run at up to system speed, can generate interrupts, and can be continuously reprogrammed via the DMA, or trigger DMA transfers to other peripherals.
- 8x PIO state machines (Chapter 3) can count 32-bit values at system speed, and generate interrupts.
- The DMA (Section 2.5) has four internal pacing timers, which trigger transfers at regular intervals.
- Each Cortex-M0+ core (Section 2.4) has a standard 24-bit SysTick timer, counting either the microsecond tick (Section 4.7.2) or the system clock.

4.6.2. Counter

The timer has a 64-bit counter, but RP2040 only has a 32-bit data bus. This means that the `TIME` value is accessed through a pair of registers. These are:

- `TIMEHW` and `TIMELW` to write the time
- `TIMEHR` and `TIMELR` to read the time

These pairs are used by accessing the lower register, `L`, followed by the higher register, `H`. In the read case, reading the `L` register latches the value in the `H` register so that an accurate time can be read. Alternatively, `TIMERAWH` and `TIMERAWL` can be used to read the raw time without any latching.

⚠ CAUTION

While it is technically possible to force a new time value by writing to the `TIMEHW` and `TIMELW` registers, programmers are discouraged from doing this. This is because the timer value is expected to be monotonically increasing by the SDK which uses it for timeouts, elapsed time etc.

4.6.3. Alarms

The timer has 4 alarms, and outputs a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64-bit counter which means they can be fired at a maximum of 2^{32} microseconds into the future. This is equivalent to:

- $2^{32} \div 10^6: \sim 4295$ seconds
- $4295 \div 60: \sim 72$ minutes

ℹ NOTE

This timer is expected to be used for short sleeps. If you want a longer alarm see Section 4.8.

To enable an alarm:

- Enable the interrupt at the timer with a write to the appropriate alarm bit in `INTE`: i.e. `(1 << 0)` for `ALARMO`
- Enable the appropriate timer interrupt at the processor (see Section 2.3.2)
- Write the time you would like the interrupt to fire to `ALARMO` (i.e. the current value in `TIMERAWL` plus your desired alarm time in microseconds). Writing the time to the `ALARM` register sets the `ARMED` bit as a side effect.

参考时钟（图28），通常直接连接至晶体振荡器（第2.16节）。

64位计数器实际上不会溢出（以1MHz计时，持续数千年），因此系统定时器在实际应用中是完全单调递增的。

4.6.1.1. RP2040上的其他定时器资源

系统定时器旨在为软件提供全局时基。RP2040具有多种其他可编程计数器资源，能够提供定期中断或触发DMA传输。

- PWM（第4.5节）包含8个16位可编程计数器，最高运行速度可达系统时钟频率，能够产生中断，且可通过DMA持续重新编程，或触发对其他外设的DMA传输。
- 8个PIO状态机（第3章）能够以系统速度计数32位值并产生中断。
- DMA（第2.5节）具有四个内部节拍计时器，可定时触发传输。
- 每个 Cortex-M0+ 核心（见第 2.4 节）均配备标准的 24 位 SysTick 定时器，可计数微秒滴答（见第 4.7.2 节）或系统时钟。

4.6.2. 计数器

该定时器拥有 64 位计数器，但 RP2040 仅具备 32 位数据总线。这意味着 **TIME** 值需通过一对寄存器访问，具体如下：

- **TIMEHW** 和 **TIMELW** 用于写入时间
- **TIMEHR** 和 **TIMELR** 用于读取时间

访问该对寄存器时，先访问低位寄存器 **L**，随后访问高位寄存器 **H**。读取时，读取 **L** 寄存器会锁存 **H** 寄存器的数值，确保时间读取的准确性。或者，可使用 **TIMERAWH** 和 **TIMERAWL** 来读取未经过锁存的原始时间。

⚠ 注意

尽管技术上可通过向 **TIMEHW** 和 **TIMELW** 寄存器写入来强制设置新的时间值，但不建议程序员这样操作。这是因为计时器数值被SDK预期为单调递增，SDK利用该值实现超时、计时等功能。

4.6.3. 报警

该计时器拥有4个报警，每个报警对应一个独立中断输出。报警基于64位计数器的低32位进行匹配，意味着报警可在 2^{32} 微秒内触发。相当于：

- $2^{32} \div 10^6$ ：约4295秒
- $4295 \div 60$ ：约72分钟

ℹ 注意

此计时器适用于短时休眠。如需更长时间报警，请参见第4.8节。

启用报警步骤如下：

- 通过向**INTE** 中对应报警位写入值来启用计时器中断：如(**1 << 0**)表示启用 **ALARM0**
- 在处理器端启用相应的计时器中断（详见第2.3.2节）
- 将希望触发中断的时间写入**ALARM0**（即当前**TIMERAWL**值加上所需的报警时间，单位为微秒）。向 **ALARM**寄存器写入时间时，会作为副作用设置**ARMED**位。

Once the alarm has fired, the **ARMED** bit will be set to **0**. To clear the latched interrupt, write a **1** to the appropriate bit in **INTR**.

4.6.4. Programmer's Model

i NOTE

The Watchdog tick (see [Section 4.7.2](#)) must be running for the timer to start counting. The SDK starts this tick as part of the platform initialisation code.

4.6.4.1. Reading the time

i NOTE

Time here refers to the number of microseconds since the timer was started, it is not a clock. For that - see [Section 4.8](#).

The simplest form of reading the 64-bit time is to read **TIMELR** followed by **TIMEHR**. However, because RP2040 has 2 cores, it is unsafe to do this if the second core is executing code that can also access the timer, or if the timer is read concurrently in an IRQ handler and in thread mode. This is because reading **TIMELR** latches the value in **TIMEHR** (i.e. stops it updating) until **TIMEHR** is read. If one core reads **TIMELR** followed by another core reading **TIMELR**, the value in **TIMEHR** isn't necessarily accurate. The example below shows the simplest form of getting the 64-bit time.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c Lines 15 - 23

```
15 // Simplest form of getting 64 bit time from the timer.
16 // It isn't safe when called from 2 cores because of the latching
17 // so isn't implemented this way in the sdk
18 static uint64_t get_time(void) {
19     // Reading low latches the high value
20     uint32_t lo = timer_hw->timelr;
21     uint32_t hi = timer_hw->timehr;
22     return ((uint64_t) hi << 32u) | lo;
23 }
```

The SDK provides a **time_us_64** function that uses a more thorough method to get the 64-bit time, which makes use of the **TIMERAWH** and **TIMERAWL** registers. The **RAW** registers don't latch, and therefore make **time_us_64** safe to call from multiple cores at once.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c Lines 57 - 73

```
57 uint64_t timer_time_us_64(timer_hw_t *timer) {
58     // Need to make sure that the upper 32 bits of the timer
59     // don't change, so read that first
60     uint32_t hi = timer->timerawh;
61     uint32_t lo;
62     do {
63         // Read the lower 32 bits
64         lo = timer->timerawl;
65         // Now read the upper 32 bits again and
66         // check that it hasn't incremented. If it has loop around
67         // and read the lower 32 bits again to get an accurate value
68         uint32_t next_hi = timer->timerawh;
69         if (hi == next_hi) break;
70         hi = next_hi;
```

闹钟触发后，ARMED位将被设置为 0。要清除锁存中断，请向INTR中的相应位写入 1。

4.6.4. 程程序员模型

① 注意

看门狗计时器（参见第4.7.2节）必须运行，定时器才能开始计数。SDK在平台初始化代码中启动该计时器。

4.6.4.1 读取时间

① 注意

此处的时间指定时器启动后经过的微秒数，非时钟时间。有关时钟时间，请参见第4.8节。

读取64位时间的最简方式是先读取TIMELR，再读取TIMEHR。但鉴于RP2040拥有双核心，若第二核心执行的代码可访问定时器，或在IRQ处理程序与线程模式中并发读取定时器，则此方法不安全。这是因为读取 TIMELR 会锁存 TIMEHR 中的值（即停止其更新），直到 TIMEHR 被读取。若一个核心读取 TIMELR，随后另一个核心也读取 TIMELR，则 TIMEHR 中的值未必准确。以下示例展示了获取 64 位时间的最简形式。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c 第 15 至 23 行

```
15 // 从计时器获取 64 位时间的最简形式。
16 // 由于锁存机制，多个核心调用时不安全
17 // 因此 SDK 中未采用此实现方式
18 static uint64_t get_time(void) {
19     // 读取低位时锁存高位值
20     uint32_t lo = timer_hw->timelr;
21     uint32_t hi = timer_hw->timehr;
22     return ((uint64_t) hi << 32u) | lo;
23 }
```

SDK 提供了一个 `time_us_64` 函数，采用更完善的方法获取64位时间，该方法利用了 TIMERAWH 和 TIMERAWL 寄存器。由于 `RAW` 寄存器不具备锁存功能，故 `ime_us_64` 函数可安全地被多个核心同时调用。

SDK：https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c 第57至73行

```
57 uint64_t timer_time_us_64(timer_hw_t *timer) {
58     // 需确保计时器高32位
59     // 不发生变化，故先读取高位
60     uint32_t hi = timer->timerawh;
61     uint32_t lo;
62     do {
63         // 读取低32位
64         lo = timer->timerawl;
65         // 现再次读取高32位，并
66         // 检查计数器是否未递增。如已递增，则循环
67         // 并再次读取低32位，以获取准确值
68         uint32_t next_hi = timer->timerawh;
69         if (hi == next_hi) break;
70         hi = next_hi;
```

```

71     } while (true);
72     return ((uint64_t) hi << 32u) | lo;
73 }
```

4.6.4.2. Set an alarm

The standalone timer example, timer_lowlevel, demonstrates how to set an alarm at a hardware level, without the additional abstraction over the timer that the SDK provides. To use these abstractions see [Section 4.6.4.4](#).

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c Lines 27 - 74

```

27 // Use alarm 0
28 #define ALARM_NUM 0
29 #define ALARM_IRQ timer_hardware_alarm_get_irq_num(timer_hw, ALARM_NUM)
30
31 // Alarm interrupt handler
32 static volatile bool alarm_fired;
33
34 static void alarm_irq(void) {
35     // Clear the alarm irq
36     hw_clear_bits(&timer_hw->intr, 1u << ALARM_NUM);
37
38     // Assume alarm 0 has fired
39     printf("Alarm IRQ fired\n");
40     alarm_fired = true;
41 }
42
43 static void alarm_in_us(uint32_t delay_us) {
44     // Enable the interrupt for our alarm (the timer outputs 4 alarm irqs)
45     hw_set_bits(&timer_hw->inte, 1u << ALARM_NUM);
46     // Set irq handler for alarm irq
47     irq_set_exclusive_handler(ALARM_IRQ, alarm_irq);
48     // Enable the alarm irq
49     irq_set_enabled(ALARM_IRQ, true);
50     // Enable interrupt in block and at processor
51
52     // Alarm is only 32 bits so if trying to delay more
53     // than that need to be careful and keep track of the upper
54     // bits
55     uint64_t target = timer_hw->timerawl + delay_us;
56
57     // Write the lower 32 bits of the target time to the alarm which
58     // will arm it
59     timer_hw->alarm[ALARM_NUM] = (uint32_t) target;
60 }
61
62 int main() {
63     stdio_init_all();
64     printf("Timer lowlevel!\n");
65
66     // Set alarm every 2 seconds
67     while (1) {
68         alarm_fired = false;
69         alarm_in_us(1000000 * 2);
70         // Wait for alarm to fire
71         while (!alarm_fired);
72     }
73 }
```

```

71     } while (true);
72     return ((uint64_t) hi << 32u) | lo;
73 }

```

4.6.4.2. 设置闹钟

独立计时器示例 timer_lowlevel 展示了如何在硬件级别设置闹钟，而无需使用 SDK 提供的计时器抽象层。有关使用这些抽象层的详细说明，请参见第 4.6.4.4 节。

Pico 示例：https://github.com/raspberrypi/pico-examples/blob/master/timer/timer_lowlevel/timer_lowlevel.c, 第 27 行至第 74 行

```

27 // 使用闹钟 0
28 #define ALARM_NUM 0
29 #define ALARM_IRQ timer_hw_alarm_get_irq_num(timer_hw, ALARM_NUM)
30
31 // 警报中断处理程序
32 static volatile bool alarm_fired;
33
34 static void alarm_irq(void) {
35     // 清除警报中断
36     hw_clear_bits(&timer_hw->intr, 1u << ALARM_NUM);
37
38     // 假设警报0已触发
39     printf("Alarm IRQ fired\n");
40     alarm_fired = true;
41 }
42
43 static void alarm_in_us(uint32_t delay_us) {
44     // 启用我们的警报中断（定时器输出4个警报中断）
45     hw_set_bits(&timer_hw->inte, 1u << ALARM_NUM);
46     // 设置警报中断处理程序
47     irq_set_exclusive_handler(ALARM_IRQ, alarm_irq);
48     // 启用警报中断
49     irq_set_enabled(ALARM_IRQ, true);
50     // 在模块和处理器处启用中断
51
52     // 警报仅为32位，如果尝试延迟超过该值，
53     // 则需谨慎并跟踪高位
54     // 位
55     uint64_t target = timer_hw->timerawl + delay_us;
56
57     // 将目标时间的低32位写入警报寄存器以启动警报
58     // will arm it
59     timer_hw->alarm[ALARM_NUM] = (uint32_t) target;
60 }
61
62 int main() {
63     stdio_init_all();
64     printf("Timer lowlevel!\n");
65
66     // 每隔2秒设置一次警报
67     while (1) {
68         alarm_fired = false;
69         alarm_in_us(1000000 * 2);
70         // 等待警报触发
71         while (!alarm_fired);
72     }
73 }

```

4.6.4.3. Busy wait

If you don't want to use an alarm to wait for a period of time, instead use a while loop. The SDK provides various `busy_wait_` functions to do this:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c Lines 77 - 122

```

77 void timer_busy_wait_us_32(timer_hw_t *timer, uint32_t delay_us) {
78     if (0 <= (int32_t)delay_us) {
79         // we only allow 31 bits, otherwise we could have a race in the loop below with
80         // values very close to 2^32
81         uint32_t start = timer->timerawl;
82         while (timer->timerawl - start < delay_us) {
83             tight_loop_contents();
84         }
85     } else {
86         busy_wait_us(delay_us);
87     }
88 }
89
90 void timer_busy_wait_us(timer_hw_t *timer, uint64_t delay_us) {
91     uint64_t base = timer_time_us_64(timer);
92     uint64_t target = base + delay_us;
93     if (target < base) {
94         target = (uint64_t)-1;
95     }
96     absolute_time_t t;
97     update_us_since_boot(&t, target);
98     timer_busy_wait_until(timer, t);
99 }
100
101 void timer_busy_wait_ms(timer_hw_t *timer, uint32_t delay_ms)
102 {
103     if (delay_ms <= 0x7fffffff / 1000) {
104         timer_busy_wait_us_32(timer, delay_ms * 1000);
105     } else {
106         timer_busy_wait_us(timer, delay_ms * 1000ull);
107     }
108 }
109
110 void timer_busy_wait_until(timer_hw_t *timer, absolute_time_t t) {
111     uint64_t target = to_us_since_boot(t);
112     uint32_t hi_target = (uint32_t)(target >> 32u);
113     uint32_t hi = timer->timerawh;
114     while (hi < hi_target) {
115         hi = timer->timerawh;
116         tight_loop_contents();
117     }
118     while (hi == hi_target && timer->timerawl < (uint32_t) target) {
119         hi = timer->timerawh;
120         tight_loop_contents();
121     }
122 }
```

4.6.4.4. Complete example using SDK

4.6.4.3. 忙等待

如果您不想使用闹钟等待一段时间，可以改用 while 循环。SDK 提供了多种 `busy_wait_` 函数用于此目的：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_timer/timer.c 第77至122行

```

77 void timer_busy_wait_us_32(timer_hw_t *timer, uint32_t delay_us) {
78     if (0 <= (int32_t)delay_us) {
79         // 仅允许使用31位，否则接近2^32的值在下面的循环中可能引发竞态条件
80         // values very close to 2^32
81         uint32_t start = timer->timerawl;
82         while (timer->timerawl - start < delay_us) {
83             tight_loop_contents();
84         }
85     } else {
86         busy_wait_us(delay_us);
87     }
88 }
89
90 void timer_busy_wait_us(timer_hw_t *timer, uint64_t delay_us) {
91     uint64_t base = timer_time_us_64(timer);
92     uint64_t target = base + delay_us;
93     if (target < base) {
94         target = (uint64_t)-1;
95     }
96     absolute_time_t t;
97     update_us_since_boot(&t, target);
98     timer_busy_wait_until(timer, t);
99 }
100
101 void timer_busy_wait_ms(timer_hw_t *timer, uint32_t delay_ms)
102 {
103     if (delay_ms <= 0x7fffffff / 1000) {
104         timer_busy_wait_us_32(timer, delay_ms * 1000);
105     } else {
106         timer_busy_wait_us(timer, delay_ms * 1000ull);
107     }
108 }
109
110 void timer_busy_wait_until(timer_hw_t *timer, absolute_time_t t) {
111     uint64_t target = to_us_since_boot(t);
112     uint32_t hi_target = (uint32_t)(target >> 32u);
113     uint32_t hi = timer->timerawh;
114     while (hi < hi_target) {
115         hi = timer->timerawh;
116         tight_loop_contents();
117     }
118     while (hi == hi_target && timer->timerawl < (uint32_t) target) {
119         hi = timer->timerawh;
120         tight_loop_contents();
121     }
122 }
```

4.6.4.4. 使用 SDK 的完整示例

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/timer/hello_timer/hello_timer.c Lines 11 - 57

```

11 volatile bool timer_fired = false;
12
13 int64_t alarm_callback(alarm_id_t id, __unused void *user_data) {
14     printf("Timer %d fired!\n", (int) id);
15     timer_fired = true;
16     // Can return a value here in us to fire in the future
17     return 0;
18 }
19
20 bool repeating_timer_callback(__unused struct repeating_timer *t) {
21     printf("Repeat at %lld\n", time_us_64());
22     return true;
23 }
24
25 int main() {
26     stdio_init_all();
27     printf("Hello Timer!\n");
28
29     // Call alarm_callback in 2 seconds
30     add_alarm_in_ms(2000, alarm_callback, NULL, false);
31
32     // Wait for alarm callback to set timer_fired
33     while (!timer_fired) {
34         tight_loop_contents();
35     }
36
37     // Create a repeating timer that calls repeating_timer_callback.
38     // If the delay is > 0 then this is the delay between the previous callback ending and the
39     // next starting.
40     // If the delay is negative (see below) then the next call to the callback will be exactly
41     // 500ms after the
42     // start of the call to the last callback
43     struct repeating_timer timer;
44     add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
45     sleep_ms(3000);
46     bool cancelled = cancel_repeating_timer(&timer);
47     printf("cancelled... %d\n", cancelled);
48     sleep_ms(2000);
49
50     // Negative delay so means we will call repeating_timer_callback, and call it again
51     // 500ms later regardless of how long the callback took to execute
52     add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
53     sleep_ms(3000);
54     cancelled = cancel_repeating_timer(&timer);
55     printf("cancelled... %d\n", cancelled);
56     sleep_ms(2000);
57     printf("Done\n");
58     return 0;
59 }
```

4.6.5. List of Registers

The Timer registers start at a base address of **0x40054000** (defined as **TIMER_BASE** in SDK).

Table 527. List of
TIMER registers

Offset	Name	Info
0x00	TIMEHW	Write to bits 63:32 of time always write timelw before timehw

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/timer/hello_timer/hello_timer.c 第11至57行

```

11 volatile bool timer_fired = false;
12
13 int64_t alarm_callback(alarm_id_t id, __unused void *user_data) {
14     printf("定时器%d触发! \n", (int) id);
15     timer_fired = true;
16     // 可返回一个微秒数值用于将来触发
17     return 0;
18 }
19
20 bool repeating_timer_callback(__unused struct repeating_timer *t) {
21     printf("重复触发时间: %lld\n", time_us_64());
22     return true;
23 }
24
25 int main() {
26     stdio_init_all();
27     printf("你好, 定时器! \n");
28
29     // 两秒后调用 alarm_callback
30     add_alarm_in_ms(2000, alarm_callback, NULL, false);
31
32     // 等待闹钟回调以设置 timer_fired
33     while (!timer_fired) {
34         tight_loop_contents();
35     }
36
37     // 创建一个重复定时器, 调用 repeating_timer_callback。
38 // 如果延迟 > 0, 则表示前一次回调结束到下一次回调开始之间的延迟。
39 // 如果延迟为负值 (详见下文), 则下一次回调将精确地在上一次回调开始 500ms 后触发
40
41     // start of the call to the last callback
42     struct repeating_timer timer;
43     add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
44     sleep_ms(3000);
45     bool cancelled = cancel_repeating_timer(&timer);
46     printf("cancelled... %d\n", cancelled);
47     sleep_ms(2000);
48
49     // 负延迟意味着我们会调用 repeating_timer_callback, 并随后再次调用
50     // 无论回调执行时间长短, 500 毫秒后触发
51     add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
52     sleep_ms(3000);
53     cancelled = cancel_repeating_timer(&timer);
54     sleep_ms(2000);
55     printf("Done\n");
56     return 0;
57 }
```

4.6.5. 寄存器列表

定时器寄存器起始地址为 `0x40054000` (在 SDK 中定义为 `TIMER_BASE`)。

表 527.
定时器寄存器列表

偏移量	名称	说明
0x00	TIMEHW	写入 time 的 63:32 位 必须先写入 timelw, 然后写入 timehw

Offset	Name	Info
0x04	TIMELW	Write to bits 31:0 of time writes do not get copied to time until timehw is written
0x08	TIMEHR	Read from bits 63:32 of time always read timelr before timehr
0x0c	TIMELR	Read from bits 31:0 of time
0x10	ALARM0	Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM0 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x14	ALARM1	Arm alarm 1, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM1 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x18	ALARM2	Arm alarm 2, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM2 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x1c	ALARM3	Arm alarm 3, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM3 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.
0x20	ARMED	Indicates the armed/disarmed status of each alarm. A write to the corresponding ALARMx register arms the alarm. Alarms automatically disarm upon firing, but writing ones here will disarm immediately without waiting to fire.
0x24	TIMERAWH	Raw read from bits 63:32 of time (no side effects)
0x28	TIMERAWL	Raw read from bits 31:0 of time (no side effects)
0x2c	DBGPAUSE	Set bits high to enable pause when the corresponding debug ports are active
0x30	PAUSE	Set high to pause the timer
0x34	INTR	Raw Interrupts
0x38	INTE	Interrupt Enable
0x3c	INTF	Interrupt Force
0x40	INTS	Interrupt status after masking & forcing

TIMER: TIMEHW Register

Offset: 0x00

偏移量	名称	说明
0x04	TIMELW	写入 time 的 31:0 位 写入操作在写入 timehw 前不会复制到 time
0x08	TIMEHR	读取 time 的 63:32 位 必须先读取 timelr，然后读取 timehr
0x0c	TIMELR	读取 time 的 31:0 位
0x10	ALARM0	使能报警器0，并配置其触发时间。 使能后，当 TIMER_ALARM0 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x14	报警器1	使能报警器1，并配置其触发时间。 使能后，当 TIMER_ALARM1 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x18	报警器2	使能报警器2，并配置其触发时间。 使能后，当 TIMER_ALARM2 == TIMELR 时报警器触发。报警 器触发后将自动解除使能，也可通过 ARMED 状 态寄存器提前解除使能。
0x1c	报警器3	使能报警器3，并配置其触发时间。 使能后，当 TIMER_ALARM3 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。
0x20	ARMED	指示各报警器的使能/解除使能状态。 向对应的 ALARMx 寄存器写入数据即使能该报警器。 报警触发后会自动解除使能，写入 1 可立即解除，无需等待触 发。
0x24	TIMERAWH	对时间的第63至32位进行原始读取（无副作用）
0x28	TIMERAWL	对时间的第31至0位进行原始读取（无副作用）
0x2c	DBGPAUSE	将对应位设为高电平以启用调试端口激活时的暂停
0x30	PAUSE	置高以暂停计时器
0x34	INTR	原始中断
0x38	INTE	中断使能
0x3c	INTF	中断强制
0x40	INTS	掩码及强制后的中断状态

TIMER: TIMEHW 寄存器

偏移: 0x00

Table 528. TIMEHW Register

Bits	Description	Type	Reset
31:0	Write to bits 63:32 of time always write timelw before timehw	WF	0x00000000

TIMER: TIMELW Register

Offset: 0x04

Table 529. TIMELW Register

Bits	Description	Type	Reset
31:0	Write to bits 31:0 of time writes do not get copied to time until timehw is written	WF	0x00000000

TIMER: TIMEHR Register

Offset: 0x08

Table 530. TIMEHR Register

Bits	Description	Type	Reset
31:0	Read from bits 63:32 of time always read timelr before timehr	RO	0x00000000

TIMER: TIMELR Register

Offset: 0x0c

Table 531. TIMELR Register

Bits	Description	Type	Reset
31:0	Read from bits 31:0 of time	RO	0x00000000

TIMER: ALARM0 Register

Offset: 0x10

Table 532. ALARM0 Register

Bits	Description	Type	Reset
31:0	Arm alarm 0, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM0 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.	RW	0x00000000

TIMER: ALARM1 Register

Offset: 0x14

Table 533. ALARM1 Register

Bits	Description	Type	Reset
31:0	Arm alarm 1, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM1 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.	RW	0x00000000

TIMER: ALARM2 Register

Offset: 0x18

Table 534. ALARM2 Register

表 528. TIMEHW
寄存器

位	描述	类型	复位值
31:0	写入 time 的 63:32 位 必须先写入 timelw，然后写入 timehw	WF	0x00000000

TIMER：TIMEHW 寄存器

偏移: 0x04

表 529. TIMELW
寄存器

位	描述	类型	复位值
31:0	写入 time 的 31:0 位 写入操作在写入 timehw 前不会复制到 time	WF	0x00000000

TIMER：TIMEHR 寄存器

偏移: 0x08

表 530. TIMEHR
寄存器

位	描述	类型	复位值
31:0	读取 time 的 63:32 位 必须先读取 timelr，然后读取 timehr	只读	0x00000000

TIMER：TIMELR 寄存器

偏移: 0x0c

表 531. TIMELR
寄存器

位	描述	类型	复位值
31:0	读取 time 的 31:0 位	只读	0x00000000

TIMER：ALARM0 寄存器

偏移: 0x10

表 532. ALARM0
寄存器

位	描述	类型	复位值
31:0	使能报警器0，并配置其触发时间。 使能后，当 TIMER_ALARM0 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM1 寄存器

偏移量: 0x14

表 533. ALARM1
寄存器

位	描述	类型	复位值
31:0	使能报警器1，并配置其触发时间。 使能后，当 TIMER_ALARM1 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM2 寄存器

偏移: 0x18

表 534. ALARM2
寄存器

Bits	Description	Type	Reset
31:0	Arm alarm 2, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM2 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.	RW	0x00000000

TIMER: ALARM3 Register

Offset: 0x1c

Table 535. ALARM3 Register

Bits	Description	Type	Reset
31:0	Arm alarm 3, and configure the time it will fire. Once armed, the alarm fires when TIMER_ALARM3 == TIMELR. The alarm will disarm itself once it fires, and can be disarmed early using the ARMED status register.	RW	0x00000000

TIMER: ARMED Register

Offset: 0x20

Table 536. ARMED Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3:0	Indicates the armed/disarmed status of each alarm. A write to the corresponding ALARMx register arms the alarm. Alarms automatically disarm upon firing, but writing ones here will disarm immediately without waiting to fire.	WC	0x0

TIMER: TIMERAWH Register

Offset: 0x24

Table 537. TIMERAWH Register

Bits	Description	Type	Reset
31:0	Raw read from bits 63:32 of time (no side effects)	RO	0x00000000

TIMER: TIMERAWL Register

Offset: 0x28

Table 538. TIMERAWL Register

Bits	Description	Type	Reset
31:0	Raw read from bits 31:0 of time (no side effects)	RO	0x00000000

TIMER: DBGPAUSE Register

Offset: 0x2c

Description

Set bits high to enable pause when the corresponding debug ports are active

Table 539. DBGPAUSE Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	DBG1: Pause when processor 1 is in debug mode	RW	0x1
1	DBG0: Pause when processor 0 is in debug mode	RW	0x1

位	描述	类型	复位值
31:0	使能报警器2，并配置其触发时间。 使能后，当 TIMER_ALARM2 == TIMELR 时报警器触发。报警器触发后将自动解除使能，也可通过 ARMED 状态寄存器提前解除使能。	读写	0x00000000

定时器：ALARM3 寄存器

偏移量: 0x1c

表 535. ALARM3 寄存器

位	描述	类型	复位值
31:0	使能报警器3，并配置其触发时间。 使能后，当 TIMER_ALARM3 == TIMELR 时报警器触发。 报警器触发后将自动解除使能，也可通过 ARME_D 状态寄存器提前解除使能。	读写	0x00000000

定时器：ARMED 寄存器

偏移量: 0x20

表 536. ARMED 寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3:0	指示各报警器的使能/解除使能状态。 向对应的 ALARMx 寄存器写入数据即使能该报警器。 报警触发后会自动解除使能，写入 1 可立即解除，无需等待触发。	WC	0x0

定时器：TIMERAWH 寄存器

偏移量: 0x24

表 537. TIMERAWH 寄存器

位	描述	类型	复位值
31:0	对时间的第63至32位进行原始读取（无副作用）	只读	0x00000000

定时器：TIMERAWL 寄存器

偏移量: 0x28

表 538. TIMERAWL 寄存器

位	描述	类型	复位值
31:0	对时间的第31至0位进行原始读取（无副作用）	只读	0x00000000

定时器：DBGPAUSE 寄存器

偏移量: 0x2c

描述

将对应位设为高电平以在相应调试端口激活时启用暂停

表 539. DBGPAUSE 寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	DBG1 : 当处理器 1 处于调试模式时暂停	读写	0x1
1	DBG0 : 当处理器0处于调试模式时暂停	读写	0x1

Bits	Description	Type	Reset
0	Reserved.	-	-

TIMER: PAUSE Register

Offset: 0x30

Table 540. PAUSE Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Set high to pause the timer	RW	0x0

TIMER: INTR Register

Offset: 0x34

Description

Raw Interrupts

Table 541. INTR Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	ALARM_3	WC	0x0
2	ALARM_2	WC	0x0
1	ALARM_1	WC	0x0
0	ALARM_0	WC	0x0

TIMER: INTE Register

Offset: 0x38

Description

Interrupt Enable

Table 542. INTE Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	ALARM_3	RW	0x0
2	ALARM_2	RW	0x0
1	ALARM_1	RW	0x0
0	ALARM_0	RW	0x0

TIMER: INTF Register

Offset: 0x3c

Description

Interrupt Force

Table 543. INTF Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	ALARM_3	RW	0x0

位	描述	类型	复位值
0	保留。	-	-

TIMER: PAUSE 寄存器

偏移量: 0x30

表540. PAUSE
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	置高以暂停计时器	读写	0x0

TIMER: INTR 寄存器

偏移量: 0x34

描述

原始中断

表541. INTR
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	WC	0x0
2	ALARM_2	WC	0x0
1	ALARM_1	WC	0x0
0	ALARM_0	WC	0x0

TIMER: INTE 寄存器

偏移量: 0x38

描述

中断使能

表542. INTE
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	读写	0x0
2	ALARM_2	读写	0x0
1	ALARM_1	读写	0x0
0	ALARM_0	读写	0x0

TIMER: INTF 寄存器

偏移量: 0x3c

描述

中断强制

表543. INTF
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	读写	0x0

Bits	Description	Type	Reset
2	ALARM_2	RW	0x0
1	ALARM_1	RW	0x0
0	ALARM_0	RW	0x0

TIMER: INTS Register

Offset: 0x40

Description

Interrupt status after masking & forcing

Table 544. INTS Register

Bits	Description	Type	Reset
31:4	Reserved.	-	-
3	ALARM_3	RO	0x0
2	ALARM_2	RO	0x0
1	ALARM_1	RO	0x0
0	ALARM_0	RO	0x0

4.7. Watchdog

4.7.1. Overview

The watchdog is a countdown timer that can restart parts of the chip if it reaches zero. This can be used to restart the processor if software gets stuck in an infinite loop. The programmer must periodically write a value to the watchdog to stop it from reaching zero.

The watchdog is reset by `rst_n_run`, which is deasserted as soon as the digital core supply (DVDD) is powered and stable, and the RUN pin is high. This allows the watchdog reset to feed into the power-on state machine (see [Section 2.13](#)) and reset controller (see [Section 2.14](#)), resetting their dependants if they are selected in the `WDSEL` register. The `WDSEL` register exists in both the power-on state machine and reset controller.

4.7.2. Tick generation

The watchdog reference clock, `clk_tick`, is driven from `clk_ref`. Ideally `clk_ref` will be configured to use the Crystal Oscillator ([Section 2.16](#)) so that it provides an accurate reference clock. The reference clock is divided internally to generate a tick (nominally 1µs) to use as the watchdog tick. The tick is configured using the `TICK` register.

NOTE

To avoid duplicating logic, this tick is also distributed to the timer (see [Section 4.6](#)) and used as the timer reference.

The SDK starts the watchdog tick in `clocks_init`:

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c Lines 16 - 18

```
16 void watchdog_start_tick(uint cycles) {
17     tick_start(TICK_WATCHDOG, cycles);
```

位	描述	类型	复位值
2	ALARM_2	读写	0x0
1	ALARM_1	读写	0x0
0	ALARM_0	读写	0x0

TIMER: INTS 寄存器

偏移: 0x40

描述

掩码及强制后的中断状态

表544. INTS
寄存器

位	描述	类型	复位值
31:4	保留。	-	-
3	ALARM_3	只读	0x0
2	ALARM_2	只读	0x0
1	ALARM_1	只读	0x0
0	ALARM_0	只读	0x0

4.7. 看门狗

4.7.1. 概述

看门狗是一种倒计时定时器，计数到零时可重启芯片的部分模块。该功能可用于在软件陷入无限循环时重启处理器。程序员必须定期向看门狗写入值，以防其计数归零。

看门狗通过 `rst_n_run` 复位，该信号在数字核心电源（DVDD）通电且稳定且RUN引脚为高电平后立即解除断言。这使得看门狗复位信号能够传入上电状态机（参见第2.13节）和复位控制器（参见第2.14节），如果它们在 `WDSEL` 寄存器中被选中，则重置其从属部分。`WDSEL` 寄存器存在于上电状态机和复位控制器中。

4.7.2. 时钟节拍生成

看门狗参考时钟 `clk_tick` 由 `clk_ref` 驱动。理想情况下，`clk_ref` 将配置为使用晶体振荡器（第2.16节），以提供精确的参考时钟。参考时钟在内部被分频以生成一个定时脉冲（标称1μs），用作看门狗时钟脉冲。该时钟脉冲通过TICK寄存器进行配置。

① 注意

为避免重复逻辑，该时钟脉冲也分发给计时器（参见第4.6节），并用作计时器参考时钟。

SDK 在 `clocks_init` 中启动看门狗计时：

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第16至18行

```
16 void watchdog_start_tick(uint cycles) {
17     tick_start(TICK_WATCHDOG, cycles);
```

18 }

4.7.3. Watchdog Counter

The watchdog counter is loaded by the [LOAD](#) register. The current value can be seen in [CTRL.TIME](#).

WARNING

Due to a logic error, the watchdog counter is decremented twice per tick. Which means the programmer needs to program double the intended count down value. The SDK examples take this issue into account. See [RP2040-E1](#) for more information.

4.7.4. Scratch Registers

The watchdog contains eight 32-bit scratch registers that can be used to store information between soft resets of the chip. A [rst_n_run](#) event triggered by toggling the RUN pin or cycling the digital core supply (DVDD) will reset the scratch registers.

The bootrom checks the watchdog scratch registers for a magic number on boot. This can be used to soft reset the chip into some user specified code. See [Section 2.8.1.1](#) for more information.

4.7.5. Programmer's Model

The SDK provides a hardware_watchdog driver to control the watchdog.

4.7.5.1. Enabling the watchdog

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c Lines 42 - 74

```

42 // Helper function used by both watchdog_enable and watchdog_reboot
43 void _watchdog_enable(uint32_t delay_ms, bool pause_on_debug) {
44     valid_params_if(HARDWARE_WATCHDOG, delay_ms <= WATCHDOG_LOAD_BITS / (1000 *
45         WATCHDOG_XFACTOR));
46     hw_clear_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
47     // Reset everything apart from ROSC and XOSC
48     hw_set_bits(&psm_hw->wdsel, PSM_WDSEL_BITS & ~(PSM_WDSEL_ROSC_BITS |
49         PSM_WDSEL_XOSC_BITS));
50     uint32_t dbg_bits = WATCHDOG_CTRL_PAUSE_DBG0_BITS |
51                         WATCHDOG_CTRL_PAUSE_DBG1_BITS |
52                         WATCHDOG_CTRL_PAUSE_JTAG_BITS;
53
54     if (pause_on_debug) {
55         hw_set_bits(&watchdog_hw->ctrl, dbg_bits);
56     } else {
57         hw_clear_bits(&watchdog_hw->ctrl, dbg_bits);
58     }
59
60     if (!delay_ms) {
61         hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_TRIGGER_BITS);
62     } else {
63         load_value = delay_ms * 1000;
64         load_value *= 2;

```

18 }

4.7.3. 看门狗计数器

看门狗计数器由 LOAD 寄存器加载。当前数值可在 CTRL.TIME 中查看。

● 警告

由于逻辑错误，看门狗计数器每个计时周期递减两次。这意味着程序员需要编写两倍于预期的倒计时值。SDK示例已考虑该问题。详见 RP2040-E1 获取更多信息。

4.7.4. 暂存寄存器

看门狗包含八个32位存储寄存器，可用于在芯片软复位期间保存信息。切换 RUN 引脚或循环数字核心电源（DVDD）触发的 `rst_n_run` 事件将重置这些存储寄存器。

启动ROM在启动时检查看门狗擦写寄存器中的魔术数字。此功能可用于软重置芯片，以运行用户指定的代码。详见第2.8.1.1节。

4.7.5. 程序员模型

SDK提供hardware_watchdog驱动程序以控制看门狗。

4.7.5.1. 启用看门狗

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第42至74行

```

42 // watchdog_enable 和 watchdog_reboot 共用的辅助函数
43 void _watchdog_enable(uint32_t delay_ms, bool pause_on_debug) {
44     valid_params_if(HARDWARE_WATCHDOG, delay_ms <= WATCHDOG_LOAD_BITS / (1000 *
45         WATCHDOG_XFACTOR));
46     hw_clear_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
47
48     // 重置除ROSC和XOSC之外的所有部分
49     hw_set_bits(&psm_hw->wdsel, PSM_WDSEL_BITS & ~(PSM_WDSEL_ROSC_BITS |
50         PSM_WDSEL_XOSC_BITS));
51
52     uint32_t dbg_bits = WATCHDOG_CTRL_PAUSE_DBG0_BITS |
53                         WATCHDOG_CTRL_PAUSE_DBG1_BITS |
54                         WATCHDOG_CTRL_PAUSE_JTAG_BITS;
55
56     if (pause_on_debug) {
57         hw_set_bits(&watchdog_hw->ctrl, dbg_bits);
58     } else {
59         hw_clear_bits(&watchdog_hw->ctrl, dbg_bits);
60     }
61
62     if (!delay_ms) {
63         hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_TRIGGER_BITS);
64     } else {
65         load_value = delay_ms * 1000;
66         load_value *= 2;
67     }
68 }
```

```

65     if (load_value > WATCHDOG_LOAD_BITS)
66         load_value = WATCHDOG_LOAD_BITS;
67
68     watchdog_update();
69
70     hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
71 }
72 }
```

4.7.5.2. Updating the watchdog counter

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c Lines 24 - 28

```

24 static uint32_t load_value;
25
26 void watchdog_update(void) {
27     watchdog_hw->load = load_value;
28 }
```

4.7.5.3. Usage

The Pico Examples repository provides a hello_watchdog example that uses the hardware_watchdog to demonstrate use of the watchdog.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/watchdog/hello_watchdog/hello_watchdog.c Lines 11 - 33

```

11 int main() {
12     stdio_init_all();
13
14     if (watchdog_enable_caused_reboot()) {
15         printf("Rebooted by Watchdog!\n");
16         return 0;
17     } else {
18         printf("Clean boot\n");
19     }
20
21     // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will
22     // reboot
23     // second arg is pause on debug which means the watchdog will pause when stepping through
24     // code
25     watchdog_enable(100, 1);
26
27     for (uint i = 0; i < 5; i++) {
28         printf("Updating watchdog %d\n", i);
29         watchdog_update();
30     }
31
32     // Wait in an infinite loop and don't update the watchdog so it reboots us
33     printf("Waiting to be rebooted by watchdog\n");
34     while(1);
35 }
```

```

65     if (load_value > WATCHDOG_LOAD_BITS)
66         load_value = WATCHDOG_LOAD_BITS;
67
68     watchdog_update();
69
70     hw_set_bits(&watchdog_hw->ctrl, WATCHDOG_CTRL_ENABLE_BITS);
71 }
72 }
```

4.7.5.2. 更新看门狗计数器

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_watchdog/watchdog.c 第24至28行

```

24 static uint32_t load_value;
25
26 void watchdog_update(void) {
27     watchdog_hw->load = load_value;
28 }
```

4.7.5.3. 使用方法

Pico 示例仓库提供了 hello_watchdog 示例，利用 hardware_watchdog 演示了看门狗的使用。

Pico 示例: https://github.com/raspberrypi/pico-examples/blob/master/watchdog/hello_watchdog/hello_watchdog.c 第11至33行

```

11 int main() {
12     stdio_init_all();
13
14     if (watchdog_enable_caused_reboot()) {
15         printf("由看门狗重启! \n");
16         return 0;
17     } else {
18         printf("Clean boot\n");
19     }
20
21 // 启用看门狗，要求每100毫秒更新一次，否则芯片将重启
22
23     // 第二个参数为调试时暂停，表示单步调试时看门狗会暂停
24     // 代码
25     watchdog_enable(100, 1);
26
27     for (uint i = 0; i < 5; i++) {
28         printf("正在更新看门狗 %d\n", i);
29         watchdog_update();
30     }
31
32     // 在无限循环中等待且不更新看门狗，导致重启
33     printf("等待看门狗重启中\n");
34     while(1);
35 }
```

4.7.6. List of Registers

The watchdog registers start at a base address of `0x40058000` (defined as `WATCHDOG_BASE` in SDK).

Table 545. List of WATCHDOG registers

Offset	Name	Info
0x00	<code>CTRL</code>	Watchdog control
0x04	<code>LOAD</code>	Load the watchdog timer.
0x08	<code>REASON</code>	Logs the reason for the last reset.
0x0c	<code>SCRATCH0</code>	Scratch register
0x10	<code>SCRATCH1</code>	Scratch register
0x14	<code>SCRATCH2</code>	Scratch register
0x18	<code>SCRATCH3</code>	Scratch register
0x1c	<code>SCRATCH4</code>	Scratch register
0x20	<code>SCRATCH5</code>	Scratch register
0x24	<code>SCRATCH6</code>	Scratch register
0x28	<code>SCRATCH7</code>	Scratch register
0x2c	<code>TICK</code>	Controls the tick generator

WATCHDOG: CTRL Register

Offset: 0x00

Description

Watchdog control

The `rst_wdsel` register determines which subsystems are reset when the watchdog is triggered.

The watchdog can be triggered in software.

Table 546. CTRL Register

Bits	Description	Type	Reset
31	TRIGGER: Trigger a watchdog reset	SC	0x0
30	ENABLE: When not enabled the watchdog timer is paused	RW	0x0
29:27	Reserved.	-	-
26	PAUSE_DBG1: Pause the watchdog timer when processor 1 is in debug mode	RW	0x1
25	PAUSE_DBG0: Pause the watchdog timer when processor 0 is in debug mode	RW	0x1
24	PAUSE_JTAG: Pause the watchdog timer when JTAG is accessing the bus fabric	RW	0x1
23:0	TIME: Indicates the number of ticks / 2 (see errata RP2040-E1) before a watchdog reset will be triggered	RO	0x000000

WATCHDOG: LOAD Register

Offset: 0x04

4.7.6. 寄存器列表

看门狗寄存器起始地址为 **0x40058000** (在SDK中定义为WATCHDOG_BASE)。

表545。 WATCH
DOG 寄存器列表

偏移量	名称	说明
0x00	CTRL	看门狗控制
0x04	LOAD	加载看门狗计时器。
0x08	REASON	记录上次复位的原因。
0x0c	SCRATCH0	暂存寄存器
0x10	SCRATCH1	暂存寄存器
0x14	SCRATCH2	暂存寄存器
0x18	SCRATCH3	暂存寄存器
0x1c	SCRATCH4	暂存寄存器
0x20	SCRATCH5	暂存寄存器
0x24	SCRATCH6	暂存寄存器
0x28	SCRATCH7	暂存寄存器
0x2c	TICK	控制滴答计时器

看门狗：CTRL寄存器

偏移: 0x00

说明

看门狗控制
rst_wdsel寄存器决定在看门狗触发时重置哪些子系统。
看门狗可由软件触发。

表 546. CTRL
寄存器

位	描述	类型	复位值
31	触发: 触发看门狗复位	SC	0x0
30	启用: 未启用时, 看门狗定时器暂停	读写	0x0
29:27	保留。	-	-
26	PAUSE_DBG1 : 当处理器1处于调试模式时暂停看门狗定时器	读写	0x1
25	PAUSE_DBG0 : 当处理器0处于调试模式时暂停看门狗定时器	读写	0x1
24	PAUSE_JTAG : 当JTAG访问总线时暂停看门狗定时器 结构	读写	0x1
23:0	TIME : 指示在触发看门狗复位前的滴答数除以2 (参见勘误 RP2040-E1)	只读	0x000000

WATCHDOG：LOAD 寄存器

偏移: 0x04

Table 547. LOAD Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:0	Load the watchdog timer. The maximum setting is 0xffffffff which corresponds to 0xffffffff / 2 ticks before triggering a watchdog reset (see errata RP2040-E1).	WF	0x000000

WATCHDOG: REASON Register

Offset: 0x08

Description

Logs the reason for the last reset. Both bits are zero for the case of a hardware reset.

Table 548. REASON Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	FORCE	RO	0x0
0	TIMER	RO	0x0

WATCHDOG: SCRATCH0, SCRATCH1, ..., SCRATCH6, SCRATCH7 Registers

Offsets: 0x0c, 0x10, ..., 0x24, 0x28

Table 549. SCRATCH0, SCRATCH1, ..., SCRATCH6, SCRATCH7 Registers

Bits	Description	Type	Reset
31:0	Scratch register. Information persists through soft reset of the chip.	RW	0x00000000

WATCHDOG: TICK Register

Offset: 0x2c

Description

Controls the tick generator

Table 550. TICK Register

Bits	Description	Type	Reset
31:20	Reserved.	-	-
19:11	COUNT: Count down timer: the remaining number clk_tick cycles before the next tick is generated.	RO	-
10	RUNNING: Is the tick generator running?	RO	-
9	ENABLE: start / stop tick generation	RW	0x1
8:0	CYCLES: Total number of clk_tick cycles before the next tick.	RW	0x000

4.8. RTC

The Real-time Clock (RTC) provides time in human-readable format and can be used to generate interrupts at specific times.

4.8.1. Storage Format

Time is stored in binary, separated in seven fields:

表547. LOAD
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:0	加载看门狗定时器。最大设置值为 0xffffffff，表示触发看门狗复位前的滴答数为 $0xffffffff / 2$ （参见勘误 RP2040-E1）。	WF	0x000000

WATCHDOG：REASON 寄存器

偏移量: 0x08

说明

记录上次复位的原因。硬件复位时，两位均为零。

表548. REASON
寄存器

位	描述	类型	复位值
31:2	保留。	-	-
1	强制	只读	0x0
0	TIMER	只读	0x0

WATCHDOG：SCRATCH0、SCRATCH1、...、SCRATCH6、SCRATCH7 寄存器

偏移量: 0x0c, 0x10, ..., 0x24, 0x28

表 549。 SCRATCH0、S
CRATCH1、...
、 SCRATCH
6、 SCRATCH7 寄存器

位	描述	类型	复位值
31:0	临时寄存器。信息在芯片软复位后仍然保留。	读写	0x00000000

看门狗：TICK 寄存器

偏移量: 0x2c

描述

控制滴答计时器

表 550。 TICK
寄存器

位	描述	类型	复位值
31:20	保留。	-	-
19:11	计数：递减计时器——下一个时钟滴答生成前剩余的 clk_tick 周期数。	只读	-
10	运行中：时钟滴答生成器是否正在运行？	只读	-
9	启用：启动 / 停止时钟滴答生成	读写	0x1
8:0	周期数：下一个时钟滴答生成前的总 clk_tick 周期数。	读写	0x000

4.8. 实时时钟 (RTC)

实时时钟 (RTC) 提供人类可读格式的时间，可用于在特定时间生成中断。

4.8.1. 存储格式

时间以二进制存储，分为七个字段：

Table 551. RTC storage format

Date/Time Field	Size	Legal values
Year	12 bits	0..4095
Month	4 bits	1..12
Day	5 bits	1..[28,29,30,31], depending on the month
Day of Week	3 bits	0..6. Sunday = 0
Hour	5 bits	0..23
Minute	6 bits	0..59
Seconds	6 bits	0..59

The RTC does not check that the programmed values are in range. Illegal values may cause unexpected behaviour.

4.8.1.1. Day of the week

Day of the week is encoded as Sun 0, Mon 1, ..., Sat 6 (i.e. ISO8601 mod 7).

There is no built-in calendar function. The RTC will not compute the correct day of the week; it will only increment the existing value.

4.8.2. Leap year

If the current value of `YEAR` in `SETUP_0` is evenly divisible by 4, a leap year is detected, and Feb 28th is followed by Feb 29th instead of March 1st. Since this is not always true (century years for example), the leap year checking can be forced off by setting `CTRL.FORCE_NOTLEAPYEAR`.

NOTE

The leap year check is done only when needed (the second following Feb 28, 23:59:59). The software can set `FORCE_NOTLEAPYEAR` anytime after 2096 Mar 1 00:00:00 as long as it arrives before 2100 Feb 28 23:59:59 (i.e. taking into account the clock domain crossing latency)

4.8.3. Interrupts

The RTC can generate an interrupt at a configured time. There is a global bit, `MATCH_ENA` in `IRQ_SETUP_0` to enable this feature, and individual enables for each time field (year, month, day, day-of-the-week, hour, minute, second). The individual enables can be used to implement repeating interrupts at specified times.

The alarm interrupt is sent to the processors and also to the ROSC and XOSC to wake them from dormant mode. See [Section 4.8.5.5](#) for more information on dormant mode.

4.8.4. Reference clock

The RTC uses a reference clock `clk_rtc`, which should be any integer frequency in the range 1...65536Hz.

The internal 1Hz reference is created by an internal clock divider which divides `clk_rtc` by an integer value. The divide value minus 1 is set in `CLKDIV_M1`.

表 551。RTC
存储格式

日期/时间字段	大小	合法值
年份	12位	0..4095
月份	4位	1..12
日期	5位	1..[28,29,30,31]，具体取决于月份
星期几	3位	0..6，星期日为0
小时	5位	0..23
分钟	6位	0..59
秒	6位	0..59

RTC不会验证所编程数值是否处于有效范围内。非法数值可能导致异常行为。

4.8.1.1. 星期几

星期几按周日为0，周一为1，...，周六为6编码（即ISO8601模7）。

无内置日历功能。RTC不会计算正确的星期几；它仅会递增现有的数值。

4.8.2. 闰年

如果SETUP_0中的 **YEAR**当前值能被4整除，则判定为闰年，2月28日之后为2月29日，而非3月1日。由于这一规则并非总是适用（例如世纪年），因此可通过设置CTRL.FORCE_NOTLEAPYEAR强制关闭闰年检测。

注意

闰年检测仅在必要时执行（即2月28日23:59:59后的第二秒）。软件可在2096年3月1日00:00:00之后任意时间设置**FORCE_NOTLEAPYEAR**，前提是必须在2100年2月28日23:59:59之前完成（考虑时钟域跨越延迟）。

4.8.3. 中断

RTC可以在设定时间生成中断。IRQ_SETUP_0中设有全局位 **MATCH_ENA**以启用该功能，并针对各时间字段（年、月、日、星期、小时、分钟、秒）设有独立的使能位。该功能可用于按照指定时间实现重复中断。

闹钟中断会发送至处理器，同时也发送至ROSC和XOSC，以唤醒它们的休眠模式。有关休眠模式的详细信息，请参见第4.8.5.5节。

4.8.4. 参考时钟

RTC使用参考时钟 **clk_rtc**，频率应为1至65536Hz范围内的任意整数。

内部1Hz参考时钟由内部时钟分频器生成，该分频器将 **clk_rtc**除以一个整数值。分频值减1的结果设置在CLKDIV_M1寄存器中。

⚠️ WARNING

While it is possible to change `CLKDIV_M1` while the RTC is enabled, it is not recommended.

`clk_rtc` can be driven either from an internal or external clock source. Those sources can be prescaled, using a fractional divider (see [Section 2.15](#)).

Examples of possible clock sources include:

- XOSC @ 12MHz / 256 = 46875Hz. To get a 1Hz reference `CLKDIV_M1` should be set to 46874.
- An external reference from a GPS, which generates one pulse per second. Configure `clk_rtc` to run from the GPIO0 clock source from GPIO pin 20. In this case, the `clk_rtc` divider is 1 and the internal RTC clock divider is also 1 (i.e. `CLKDIV_M1` = 0).

ℹ️ NOTE

All RTC register reads and writes are done from the processor clock domain `clk_sys`. All data are synchronised back and forth between the domains. Writing to the RTC will take 2 `clk_rtc` clock periods to arrive, additional to the `clk_sys` domain. This should be taken into account especially when the reference is slow (e.g. 1Hz).

4.8.5. Programmer's Model

There are three setup tasks:

- Set the 1 sec reference
- Set the clock
- Set an alarm

4.8.5.1. Configuring the 1 second reference clock:

Select the source for `clk_rtc`. This is done outside the RTC registers (see [Section 4.8.4](#)).

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c Lines 22 - 39

```

22 void rtc_init(void) {
23     // Get clk_rtc freq and make sure it is running
24     uint rtc_freq = clock_get_hz(clk_rtc);
25     assert(rtc_freq != 0);
26
27     // Take rtc out of reset now that we know clk_rtc is running
28     reset_unreset_block_num_wait_blocking(RESET_RTC);
29
30     // Set up the 1 second divider.
31     // If rtc_freq is 400 then clkdiv_m1 should be 399
32     rtc_freq -= 1;
33
34     // Check the freq is not too big to divide
35     assert(rtc_freq <= RTC_CLKDIV_M1_BITS);
36
37     // Write divide value
38     rtc_hw->clkdiv_m1 = rtc_freq;
39 }
```

● 警告

虽然可以在RTC启用时更改CLKDIV_M1，但不建议这样操作。

`clk_rtc`可由内部或外部时钟源驱动。这些时钟源可通过分数分频器进行预分频（参见第2.15节）。

可能的时钟源示例包括：

- XOSC @ 12MHz / 256 = 46875Hz。要获得1Hz参考信号，应将CLKDIV_M1设置为46874。
- 来自GPS的外部参考信号，每秒产生一个脉冲。配置 `clk_rtc`以使用GPIO引脚20的GPIO0时钟源。在此情况下，`clk_rtc`分频器为1，内部RTC时钟分频器也为1（即CLKDIV_M1 = 0）。

i 注意

所有RTC寄存器的读写均在处理器时钟域 `clk_sys`内完成。所有数据均在不同时钟域间双向同步。写入RTC需经过2个 `clk_rtc`时钟周期方可生效，此外还涉及 `clk_sys`时钟域。尤其在参考信号频率较低时（例如1Hz），应予以考虑。

4.8.5. 程序员模型

有三项设置任务：

- 设置1秒参考信号
- 设置时钟
- 设置闹钟

4.8.5.1. 配置1秒参考时钟：

选择 `clk_rtc` 的时钟源。该操作在RTC寄存器之外完成（参见第4.8.4节）。

SDK： https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第22至39行

```

22 void rtc_init(void) {
23     // 获取clk_rtc频率并确认其正在运行
24     uint rtc_freq = clock_get_hz(clk_rtc);
25     assert(rtc_freq != 0);
26
27     // 确认clk_rtc运行后解除rtc复位
28     reset_unreset_block_num_wait_blocking(RESET_RTC);
29
30     // 配置1秒分频器。
31     // 若rtc_freq为400，则clkdiv_m1应为399
32     rtc_freq -= 1;
33
34     // 检查频率是否未超过可分配最大值
35     assert(rtc_freq <= RTC_CLKDIV_M1_BITS);
36
37     // 写入分频值
38     rtc_hw->clkdiv_m1 = rtc_freq;
39 }
```

4.8.5.2. Setting up the clock

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c Lines 54 - 85

```

54 bool rtc_set_datetime(const datetime_t *t) {
55     if (!valid_datetime(t)) {
56         return false;
57     }
58
59     // Disable RTC
60     rtc_hw->ctrl = 0;
61     // Wait while it is still active
62     while (rtc_running()) {
63         tight_loop_contents();
64     }
65
66     // Write to setup registers
67     rtc_hw->setup_0 = (((uint32_t)t->year) << RTC_SETUP_0_YEAR_LSB) |
68                     (((uint32_t)t->month) << RTC_SETUP_0_MONTH_LSB) |
69                     (((uint32_t)t->day) << RTC_SETUP_0_DAY_LSB);
70     rtc_hw->setup_1 = (((uint32_t)t->dotw) << RTC_SETUP_1_DOTW_LSB) |
71                     (((uint32_t)t->hour) << RTC_SETUP_1_HOUR_LSB) |
72                     (((uint32_t)t->min) << RTC_SETUP_1_MIN_LSB) |
73                     (((uint32_t)t->sec) << RTC_SETUP_1_SEC_LSB);
74
75     // Load setup values into rtc clock domain
76     rtc_hw->ctrl = RTC_CTRL_LOAD_BITS;
77
78     // Enable RTC and wait for it to be running
79     rtc_hw->ctrl = RTC_CTRL_RTC_ENABLE_BITS;
80     while (!rtc_running()) {
81         tight_loop_contents();
82     }
83
84     return true;
85 }
```

NOTE

It is possible to change the current time while the RTC is running. Write the desired values, then set the LOAD bit in the [CTRL](#) register.

4.8.5.3. Reading the current time

The RTC time is stored across two 32-bit registers. To ensure a consistent value, [RTC_0](#) should be read before [RTC_1](#). Reading [RTC_0](#) latches the value of [RTC_1](#).

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c Lines 87 - 106

```

87 bool rtc_get_datetime(datetime_t *t) {
88     // Make sure RTC is running
89     if (!rtc_running()) {
90         return false;
91     }
92
93     // Note: RTC_0 should be read before RTC_1
94     uint32_t rtc_0 = rtc_hw->rtc_0;
95     uint32_t rtc_1 = rtc_hw->rtc_1;
```

4.8.5.2. 设置时钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第54至85行

```

54     bool rtc_set_datetime(const datetime_t *t) {
55         if (!valid_datetime(t)) {
56             return false;
57         }
58
59         // 禁用RTC
60         rtc_hw->ctrl = 0;
61         // 等待其仍处于活动状态
62         while (rtc_running()) {
63             tight_loop_contents();
64         }
65
66         // 写入设置寄存器
67         rtc_hw->setup_0 = (((uint32_t)t->year) << RTC_SETUP_0_YEAR_LSB) |
68             (((uint32_t)t->month) << RTC_SETUP_0_MONTH_LSB) |
69             (((uint32_t)t->day) << RTC_SETUP_0_DAY_LSB);
70         rtc_hw->setup_1 = (((uint32_t)t->dotw) << RTC_SETUP_1_DOTW_LSB) |
71             (((uint32_t)t->hour) << RTC_SETUP_1_HOUR_LSB) |
72             (((uint32_t)t->min) << RTC_SETUP_1_MIN_LSB) |
73             (((uint32_t)t->sec) << RTC_SETUP_1_SEC_LSB);
74
75         // 将设置值加载至 rtc 时钟域
76         rtc_hw->ctrl = RTC_CTRL_LOAD_BITS;
77
78         // 启用 RTC 并等待其正常运行
79         rtc_hw->ctrl = RTC_CTRL_RTC_ENABLE_BITS;
80         while (!rtc_running()) {
81             tight_loop_contents();
82         }
83
84         return true;
85     }

```

① 注意

在 RTC 运行期间，当前时间可以被更改。写入所需数值后，在 CTRL 寄存器中设置 LOAD 位。

4.8.5.3. 读取当前时间

RTC 时间存储于两个 32 位寄存器中。为确保数值一致，应先读取 RTC_0，再读取 RTC_1。读取 RTC_0 会锁存 RTC_1 的值。

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第87至106行

```

87     bool rtc_get_datetime(datetime_t *t) {
88         // 确保 RTC 正常运行
89         if (!rtc_running()) {
90             return false;
91         }
92
93         // 注意：RTC_0 应先于 RTC_1 读取
94         uint32_t rtc_0 = rtc_hw->rtc_0;
95         uint32_t rtc_1 = rtc_hw->rtc_1;

```

```

96
97     t->dotw  = (int8_t) (((rtc_0 & RTC_RTC_0_DOTW_BITS ) >> RTC_RTC_0_DOTW_LSB));
98     t->hour  = (int8_t) (((rtc_0 & RTC_RTC_0_HOUR_BITS ) >> RTC_RTC_0_HOUR_LSB));
99     t->min   = (int8_t) (((rtc_0 & RTC_RTC_0_MIN_BITS ) >> RTC_RTC_0_MIN_LSB));
100    t->sec   = (int8_t) (((rtc_0 & RTC_RTC_0_SEC_BITS ) >> RTC_RTC_0_SEC_LSB));
101    t->year  = (int16_t) (((rtc_1 & RTC_RTC_1_YEAR_BITS ) >> RTC_RTC_1_YEAR_LSB));
102    t->month = (int8_t) (((rtc_1 & RTC_RTC_1_MONTH_BITS) >> RTC_RTC_1_MONTH_LSB));
103    t->day   = (int8_t) (((rtc_1 & RTC_RTC_1_DAY_BITS ) >> RTC_RTC_1_DAY_LSB));
104
105    return true;
106 }

```

4.8.5.4. Configuring an Alarm

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/p2_common/hardware_rtc/rtc.c Lines 146 - 182

```

146 void rtc_set_alarm(const datetime_t *t, rtc_callback_t user_callback) {
147     rtc_disable_alarm();
148
149     // Only add to setup if it isn't -1
150     rtc_hw->irq_setup_0 = ((t->year < 0) ? 0 : (((uint32_t)t->year) <<
151                                         RTC_IRQ_SETUP_0_YEAR_LSB)) |
152                                         (((t->month < 0) ? 0 : (((uint32_t)t->month) <<
153                                         RTC_IRQ_SETUP_0_MONTH_LSB)) |
154                                         (((t->day < 0) ? 0 : (((uint32_t)t->day) <<
155                                         RTC_IRQ_SETUP_0_DAY_LSB)));
156     rtc_hw->irq_setup_1 = ((t->dotw < 0) ? 0 : (((uint32_t)t->dotw) <<
157                                         RTC_IRQ_SETUP_1_DOTW_LSB)) |
158                                         (((t->hour < 0) ? 0 : (((uint32_t)t->hour) <<
159                                         RTC_IRQ_SETUP_1_HOUR_LSB)) |
160                                         (((t->min < 0) ? 0 : (((uint32_t)t->min) <<
161                                         RTC_IRQ_SETUP_1_MIN_LSB)) |
162                                         (((t->sec < 0) ? 0 : (((uint32_t)t->sec) <<
163                                         RTC_IRQ_SETUP_1_SEC_LSB));
164
165     // Set the match enable bits for things we care about
166     if (t->year >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_YEAR_ENA_BITS);
167     if (t->month >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_MONTH_ENA_BITS);
168     if (t->day >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_DAY_ENA_BITS);
169     if (t->dotw >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_DOTW_ENA_BITS);
170     if (t->hour >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_HOUR_ENA_BITS);
171     if (t->min >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_MIN_ENA_BITS);
172     if (t->sec >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_SEC_ENA_BITS);
173
174     // Does it repeat? I.e. do we not match on any of the bits
175     _alarm_repeats = rtc_alarm_repeats(t);
176
177     // Store function pointer we can call later
178     _callback = user_callback;
179
180     irq_set_exclusive_handler(RTC_IRQ, rtc_irq_handler);
181
182     // Enable the IRQ at the peri
183     rtc_hw->inte = RTC_INTE_RTC_BITS;
184
185     // Enable the IRQ at the proc
186     irq_set_enabled(RTC_IRQ, true);
187
188     rtc_enable_alarm();

```

```

96
97     t->dotw  = (int8_t) (((rtc_0 & RTC_RTC_0_DOTW_BITS ) >> RTC_RTC_0_DOTW_LSB);
98     t->hour  = (int8_t) (((rtc_0 & RTC_RTC_0_HOUR_BITS ) >> RTC_RTC_0_HOUR_LSB);
99     t->min   = (int8_t) (((rtc_0 & RTC_RTC_0_MIN_BITS ) >> RTC_RTC_0_MIN_LSB);
100    t->sec   = (int8_t) (((rtc_0 & RTC_RTC_0_SEC_BITS ) >> RTC_RTC_0_SEC_LSB);
101    t->year  = (int16_t) (((rtc_1 & RTC_RTC_1_YEAR_BITS ) >> RTC_RTC_1_YEAR_LSB));
102    t->month = (int8_t) (((rtc_1 & RTC_RTC_1_MONTH_BITS) >> RTC_RTC_1_MONTH_LSB));
103    t->day   = (int8_t) (((rtc_1 & RTC_RTC_1_DAY_BITS ) >> RTC_RTC_1_DAY_LSB));
104
105    return true;
106 }

```

4.8.5.4. 配置闹钟

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_rtc/rtc.c 第146至182行

```

146 void rtc_set_alarm(const datetime_t *t, rtc_callback_t user_callback) {
147     rtc_disable_alarm();
148
149     // 仅当值不为 -1 时加入设置
150     rtc_hw->irq_setup_0 = ((t->year < 0) ? 0 : (((uint32_t)t->year) <<
151                                         RTC_IRQ_SETUP_0_YEAR_LSB)) |
152                                         (((t->month < 0) ? 0 : (((uint32_t)t->month) <<
153                                         RTC_IRQ_SETUP_0_MONTH_LSB)) |
154                                         (((t->day < 0) ? 0 : (((uint32_t)t->day) <<
155                                         RTC_IRQ_SETUP_0_DAY_LSB)));
156     rtc_hw->irq_setup_1 = ((t->dotw < 0) ? 0 : (((uint32_t)t->dotw) <<
157                                         RTC_IRQ_SETUP_1_DOTW_LSB)) |
158                                         (((t->hour < 0) ? 0 : (((uint32_t)t->hour) <<
159                                         RTC_IRQ_SETUP_1_HOUR_LSB)) |
160                                         (((t->min < 0) ? 0 : (((uint32_t)t->min) <<
161                                         RTC_IRQ_SETUP_1_MIN_LSB)) |
162                                         (((t->sec < 0) ? 0 : (((uint32_t)t->sec) <<
163                                         RTC_IRQ_SETUP_1_SEC_LSB)));
164
165     // 设置我们关注项目的匹配使能位
166     if (t->year >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_YEAR_ENA_BITS);
167     if (t->month >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_MONTH_ENA_BITS);
168     if (t->day >= 0) hw_set_bits(&rtc_hw->irq_setup_0, RTC_IRQ_SETUP_0_DAY_ENA_BITS);
169     if (t->dotw >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_DOTW_ENA_BITS);
170     if (t->hour >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_HOUR_ENA_BITS);
171     if (t->min >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_MIN_ENA_BITS);
172     if (t->sec >= 0) hw_set_bits(&rtc_hw->irq_setup_1, RTC_IRQ_SETUP_1_SEC_ENA_BITS);
173
174     // 它会重复吗? 即我们是否不匹配任何位
175     _alarm_repeats = rtc_alarm_repeats(t);
176
177     // 存储稍后可调用的函数指针
178     _callback = user_callback;
179
180     irq_set_exclusive_handler(RTC_IRQ, rtc_irq_handler);
181
182     // 启用外围设备的中断 (IRQ)
183     rtc_hw->inte = RTC_INTE_RTC_BITS;
184
185     // 启用处理器中的 IRQ
186     irq_set_enabled(RTC_IRQ, true);
187
188     rtc_enable_alarm();

```

182 }

NOTE

Recurring alarms can be created by using fewer enable bits when setting up the alarm interrupt. For example, if you only matched on seconds and the second was configured as 54 then the alarm interrupt would fire once a minute when the second was 54.

4.8.5.5. Interaction with Dormant / Sleep mode

RP2040 supports two power saving levels:

- Sleep mode, where the processors are asleep and the unused clocks in the chip are stopped (see [Section 2.15.3.5](#))
- Dormant mode, where all clocks in the chip are stopped

The RTC can wake the chip up from both of these modes. In sleep mode, RP2040 can be configured such that only `clk_rtc` (a slow RTC reference clock) is running, as well as a small amount of logic that allows the processor to wake back up. The processor is woken from sleep mode when the RTC alarm interrupt fires. See [Section 2.11.5.1](#) for more information.

To wake the chip from dormant mode:

- the RTC must be configured to use an external reference clock (supplied by a GPIO pin)
- Set up the RTC to run on an external reference
- If the processor is running off the PLL, change it to run from XOSC/ROSC
- Turn off the PLLs
- Set up the RTC with the desired wake up time (one off, or recurring)
- (optionally) power down most memories
- Invoke DORMANT mode (see [Section 2.16](#), [Section 2.17](#), and [Section 2.11.5.2](#) for more information)

4.8.6. List of Registers

The RTC registers start at a base address of `0x4005c000` (defined as `RTC_BASE` in SDK).

Table 552. List of RTC registers

Offset	Name	Info
0x00	CLKDIV_M1	Divider minus 1 for the 1 second counter. Safe to change the value when RTC is not enabled.
0x04	SETUP_0	RTC setup register 0
0x08	SETUP_1	RTC setup register 1
0x0c	CTRL	RTC Control and status
0x10	IRQ_SETUP_0	Interrupt setup register 0
0x14	IRQ_SETUP_1	Interrupt setup register 1
0x18	RTC_1	RTC register 1.
0x1c	RTC_0	RTC register 0 Read this before RTC 1!
0x20	INTR	Raw Interrupts
0x24	INTE	Interrupt Enable

182 }

i 注意

通过在设置闹钟时使用较少的启用位，可以创建周期性闹钟。例如，如果只匹配秒数且秒设置为 54，则当秒数为 54 时，闹钟中断将每分钟触发一次。

4.8.5.5. 与休眠模式的交互

RP2040 支持两个节能等级：

- 休眠模式，处理器处于休眠状态，芯片中未使用的时钟停止（参见第 2.15.3.5 节）
- 休眠模式，芯片中所有时钟均已停止

RTC 可从这两种模式中唤醒芯片。在休眠模式下，RP2040 可配置为仅运行 `clk_rtc`（一种低速 RTC 参考时钟）及少量允许处理器唤醒的逻辑。当 RTC 闹钟中断触发时，处理器将从休眠模式唤醒。详情请参见第 2.11.5.1 节。

从休眠模式唤醒芯片的方法：

- RTC 必须配置为使用外部参考时钟（由 GPIO 引脚提供）
- 设置 RTC 以使用外部参考时钟运行
- 如果处理器当前由 PLL 供电，需切换为 XOSC/ROSC 供电
- 关闭所有 PLL
- 配置 RTC 为预定的唤醒时间（单次或周期性）
- （可选）关闭大部分存储器电源
- 进入休眠模式（详见第 2.16 节、第 2.17 节及第 2.11.5.2 节）

4.8.6. 寄存器列表

RTC 寄存器起始地址为 `0x4005c000`（在 SDK 中定义为 `RTC_BASE`）。

表 552. RTC 寄存器列表

偏移量	名称	说明
0x00	<code>CLKDIV_M1</code>	1 秒计数器的分频器值减 1。RTC 未启用时，可以安全修改该值。
0x04	<code>SETUP_0</code>	RTC 配置寄存器 0
0x08	<code>SETUP_1</code>	RTC 设置寄存器 1
0x0c	<code>CTRL</code>	RTC 控制与状态
0x10	<code>IRQ_SETUP_0</code>	中断设置寄存器 0
0x14	<code>IRQ_SETUP_1</code>	中断设置寄存器 1
0x18	<code>RTC_1</code>	RTC 寄存器 1
0x1c	<code>RTC_0</code>	RTC 寄存器 0 使用 RTC 1 之前请先阅读本说明！
0x20	<code>INTR</code>	原始中断
0x24	<code>INTE</code>	中断使能

Offset	Name	Info
0x28	INTF	Interrupt Force
0x2c	INTS	Interrupt status after masking & forcing

RTC: CLKDIV_M1 Register

Offset: 0x00

Table 553. CLKDIV_M1 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	Divider minus 1 for the 1 second counter. Safe to change the value when RTC is not enabled.	RW	0x0000

RTC: SETUP_0 Register

Offset: 0x04

Description

RTC setup register 0

Table 554. SETUP_0 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:12	YEAR : Year	RW	0x000
11:8	MONTH : Month (1..12)	RW	0x0
7:5	Reserved.	-	-
4:0	DAY : Day of the month (1..31)	RW	0x00

RTC: SETUP_1 Register

Offset: 0x08

Description

RTC setup register 1

Table 555. SETUP_1 Register

Bits	Description	Type	Reset
31:27	Reserved.	-	-
26:24	DOTW : Day of the week: 1-Monday...0-Sunday ISO 8601 mod 7	RW	0x0
23:21	Reserved.	-	-
20:16	HOUR : Hours	RW	0x00
15:14	Reserved.	-	-
13:8	MIN : Minutes	RW	0x00
7:6	Reserved.	-	-
5:0	SEC : Seconds	RW	0x00

RTC: CTRL Register

Offset: 0x0c

偏移量	名称	说明
0x28	INTF	中断强制
0x2c	INTS	掩码及强制后的中断状态

RTC: CLKDIV_M1 寄存器

偏移: 0x00

表 553. CLKDIV_M1
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	1秒计数器的分频器值减1。 RTC未启用时，可以安全修改该值。	读写	0x0000

RTC: SETUP_0 寄存器

偏移: 0x04

描述

RTC配置寄存器0

表 554. SETUP_0
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	YEAR : 年	读写	0x000
11:8	MONTH : 月 (1..12)	读写	0x0
7:5	保留。	-	-
4:0	DAY : 月内日期 (1..31)	读写	0x00

RTC: SETUP_1 寄存器

偏移: 0x08

描述

RTC 设置寄存器 1

表 555. SETUP_1
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26:24	DOTW : 星期几，1-星期一...0-星期日，ISO 8601 模 7	读写	0x0
23:21	保留。	-	-
20:16	HOUR : 小时	读写	0x00
15:14	保留。	-	-
13:8	MIN : 分钟	读写	0x00
7:6	保留。	-	-
5:0	SEC : 秒	读写	0x00

RTC: CTRL 寄存器

偏移: 0x0c

Description

RTC Control and status

Table 556. CTRL Register

Bits	Description	Type	Reset
31:9	Reserved.	-	-
8	FORCE_NOTLEAPYEAR : If set, leapyear is forced off. Useful for years divisible by 100 but not by 400	RW	0x0
7:5	Reserved.	-	-
4	LOAD : Load RTC	SC	0x0
3:2	Reserved.	-	-
1	RTC_ACTIVE : RTC enabled (running)	RO	-
0	RTC_ENABLE : Enable RTC	RW	0x0

RTC: IRQ_SETUP_0 Register

Offset: 0x10

Description

Interrupt setup register 0

Table 557. IRQ_SETUP_0 Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29	MATCH_ACTIVE	RO	-
28	MATCH_ENA : Global match enable. Don't change any other value while this one is enabled	RW	0x0
27	Reserved.	-	-
26	YEAR_ENA : Enable year matching	RW	0x0
25	MONTH_ENA : Enable month matching	RW	0x0
24	DAY_ENA : Enable day matching	RW	0x0
23:12	YEAR : Year	RW	0x000
11:8	MONTH : Month (1..12)	RW	0x0
7:5	Reserved.	-	-
4:0	DAY : Day of the month (1..31)	RW	0x00

RTC: IRQ_SETUP_1 Register

Offset: 0x14

Description

Interrupt setup register 1

Table 558. IRQ_SETUP_1 Register

Bits	Description	Type	Reset
31	DOTW_ENA : Enable day of the week matching	RW	0x0
30	HOUR_ENA : Enable hour matching	RW	0x0
29	MIN_ENA : Enable minute matching	RW	0x0
28	SEC_ENA : Enable second matching	RW	0x0

说明

RTC 控制与状态

表 556. CTRL
寄存器

位	描述	类型	复位值
31:9	保留。	-	-
8	FORCE_NOTLEAPYEAR : 若设置，则强制非闰年。 适用于能被100整除但不能被400整除的年份	读写	0x0
7:5	保留。	-	-
4	LOAD : 载入 RTC	SC	0x0
3:2	保留。	-	-
1	RTC_ACTIVE : RTC 已启用（运行中）	只读	-
0	RTC_ENABLE : 启用 RTC	读写	0x0

RTC: IRQ_SETUP_0 寄存器

偏移: 0x10

描述

中断设置寄存器 0

表 557.
IRQ_SETUP_0 寄存器

位	描述	类型	复位值
31:30	保留。	-	-
29	MATCH_ACTIVE	只读	-
28	MATCH_ENA : 全局匹配使能。启用此项时请勿更改其他任何值	读写	0x0
27	保留。	-	-
26	YEAR_ENA : 启用年匹配	读写	0x0
25	MONTH_ENA : 启用月匹配	读写	0x0
24	DAY_ENA : 启用日匹配	读写	0x0
23:12	YEAR : 年	读写	0x000
11:8	MONTH : 月 (1..12)	读写	0x0
7:5	保留。	-	-
4:0	DAY : 月内日期 (1..31)	读写	0x00

RTC: IRQ_SETUP_1 寄存器

偏移量: 0x14

描述

中断设置寄存器 1

表 558.
IRQ_SETUP_1 寄存器

位	描述	类型	复位值
31	DOTW_ENA : 启用星期匹配	读写	0x0
30	HOUR_ENA : 启用小时匹配	读写	0x0
29	MIN_ENA : 启用分钟匹配	读写	0x0
28	SEC_ENA : 启用秒匹配	读写	0x0

Bits	Description	Type	Reset
27	Reserved.	-	-
26:24	DOTW: Day of the week	RW	0x0
23:21	Reserved.	-	-
20:16	HOUR: Hours	RW	0x00
15:14	Reserved.	-	-
13:8	MIN: Minutes	RW	0x00
7:6	Reserved.	-	-
5:0	SEC: Seconds	RW	0x00

RTC: RTC_1 Register

Offset: 0x18

Description

RTC register 1.

Table 559. RTC_1 Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:12	YEAR: Year	RO	-
11:8	MONTH: Month (1..12)	RO	-
7:5	Reserved.	-	-
4:0	DAY: Day of the month (1..31)	RO	-

RTC: RTC_0 Register

Offset: 0x1c

Description

RTC register 0

Read this before RTC 1!

Table 560. RTC_0 Register

Bits	Description	Type	Reset
31:27	Reserved.	-	-
26:24	DOTW: Day of the week	RF	-
23:21	Reserved.	-	-
20:16	HOUR: Hours	RF	-
15:14	Reserved.	-	-
13:8	MIN: Minutes	RF	-
7:6	Reserved.	-	-
5:0	SEC: Seconds	RF	-

RTC: INTR Register

Offset: 0x20

位	描述	类型	复位值
27	保留。	-	-
26:24	DOTW: 星期	读写	0x0
23:21	保留。	-	-
20:16	HOUR: 小时	读写	0x00
15:14	保留。	-	-
13:8	MIN: 分钟	读写	0x00
7:6	保留。	-	-
5:0	SEC: 秒	读写	0x00

RTC: RTC_1 寄存器

偏移: 0x18

描述

RTC 寄存器 1

表 559。RTC_1
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:12	YEAR: 年	只读	-
11:8	MONTH: 月 (1..12)	只读	-
7:5	保留。	-	-
4:0	DAY: 月内日期 (1..31)	只读	-

RTC: RTC_0 寄存器

偏移量: 0x1c

描述

RTC 寄存器 0

使用 RTC 1 之前请先阅读本说明!

表 560。RTC_0
寄存器

位	描述	类型	复位值
31:27	保留。	-	-
26:24	DOTW: 星期	RF	-
23:21	保留。	-	-
20:16	HOUR: 小时	RF	-
15:14	保留。	-	-
13:8	MIN: 分钟	RF	-
7:6	保留。	-	-
5:0	SEC: 秒	RF	-

RTC: INTR 寄存器

偏移量: 0x20

Description

Raw Interrupts

Table 561. INTR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	RTC	RO	0x0

RTC: INTE Register

Offset: 0x24

Description

Interrupt Enable

Table 562. INTE Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	RTC	RW	0x0

RTC: INTF Register

Offset: 0x28

Description

Interrupt Force

Table 563. INTF Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	RTC	RW	0x0

RTC: INTS Register

Offset: 0x2c

Description

Interrupt status after masking & forcing

Table 564. INTS Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	RTC	RO	0x0

4.9. ADC and Temperature Sensor

RP2040 has an internal analogue-digital converter (ADC) with the following features:

- SAR ADC (see [Section 4.9.2](#))
- 500ksps (using an independent 48MHz clock)
- 12-bit with 8.7 ENOB (see [Section 4.9.3](#))
- Five input mux:
 - Four inputs that are available on package pins shared with GPIO[29:26]

说明

原始中断

表 561。INTR 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	只读	0x0

RTC：INTE 寄存器

偏移: 0x24

说明

中断使能

表 562。INTE 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	读写	0x0

RTC：INTF 寄存器

偏移: 0x28

说明

中断强制

表 563。INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	读写	0x0

RTC：INTS 寄存器

偏移量: 0x2c

描述

掩码及强制后的中断状态

表 564。INTS 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	RTC	只读	0x0

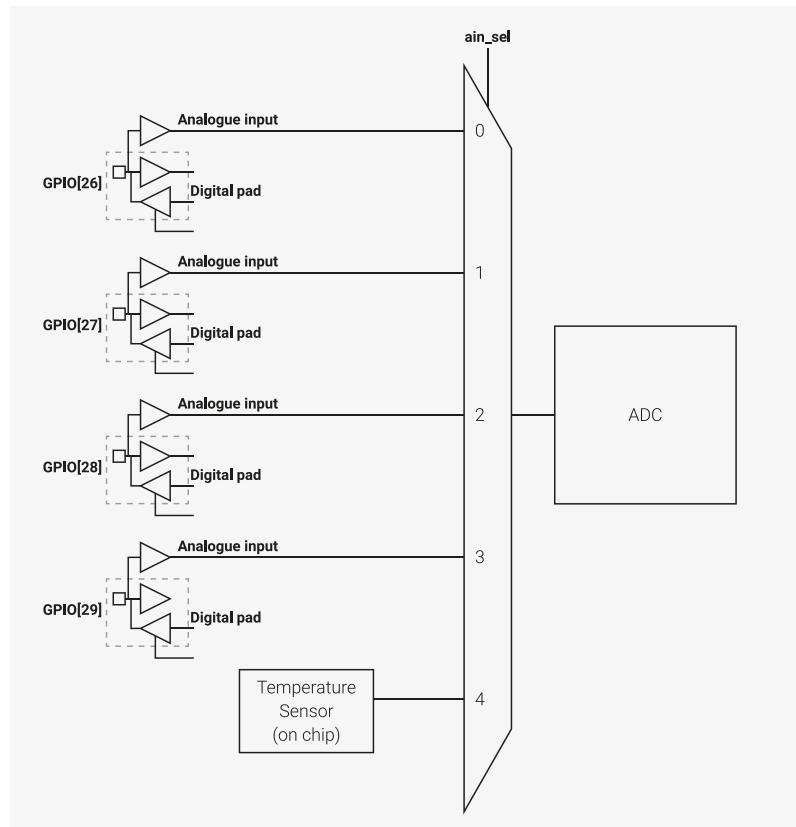
4.9. 模数转换器 (ADC) 及温度传感器

RP2040 内置模拟数字转换器 (ADC)，具备以下特性：

- 逐次逼近寄存器 (SAR) ADC (参见第 4.9.2 节)
- 500ksps (采用独立 48MHz 时钟)
- 12 位，具有 8.7 有效位数 (ENOB) (参见第 4.9.3 节)
- 五路输入复用器：
 - 四个输入通过封装引脚共享 GPIO[29:26]

- One input is dedicated to the internal temperature sensor (see [Section 4.9.5](#))
- Eight element receive sample FIFO
- Interrupt generation
- DMA interface (see [Section 4.9.2.5](#))

Figure 114. ADC Connection Diagram



NOTE

When using an ADC input shared with a GPIO pin, the pin's digital functions must be disabled by setting `IE` low and `OD` high in the pin's pad control register. See [Section 2.19.6.3, "Pad Control - User Bank"](#) for details. The maximum ADC input voltage is determined by the digital IO supply voltage (IOVDD), not the ADC supply voltage (ADC_AVDD). For example, if IOVDD is powered at 1.8V, the voltage on the ADC inputs should not exceed 1.8V even if ADC_AVDD is powered at 3.3V. Voltages greater than IOVDD will result in leakage currents through the ESD protection diodes. See [Section 5.5.3, "Pin Specifications"](#) for details.

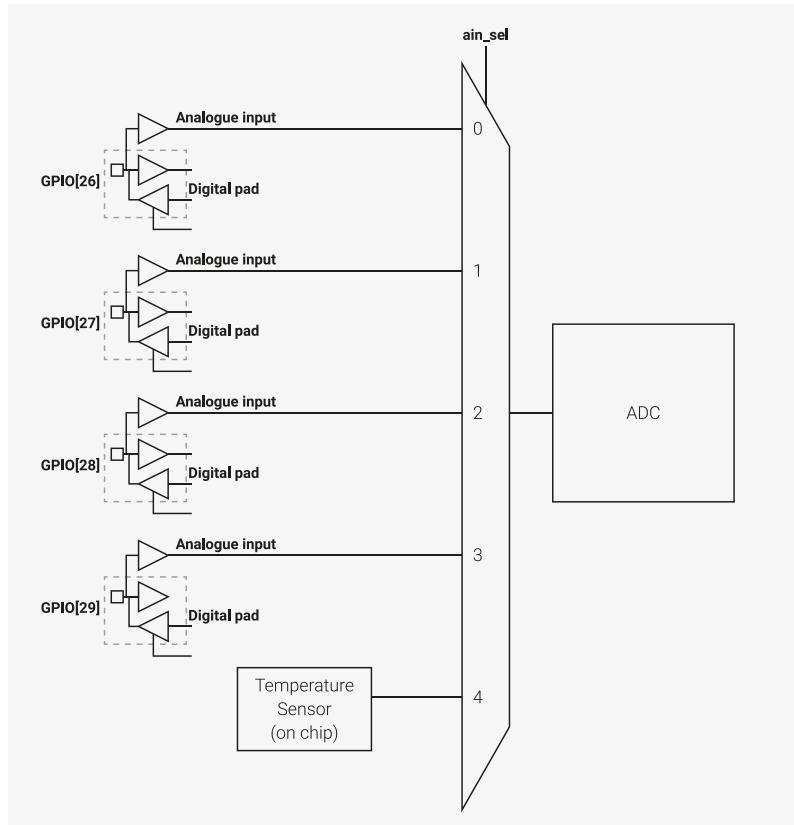
4.9.1. ADC controller

A digital controller manages the details of operating the RP2040 ADC, and provides additional functionality:

- One-shot or free-running capture mode
- Sample FIFO with DMA interface
- Pacing timer (16 integer bits, 8 fractional bits) for setting free-running sample rate
- Round-robin sampling of multiple channels in free-running capture mode
- Optional right-shift to 8 bits in free-running capture mode, so samples can be DMA'd to a byte buffer in system memory

- 一路输入专用用于内部温度传感器（参见第 4.9.5 节）
- 八元素采样接收 FIFO
- 中断生成
- DMA 接口（参见第 4.9.2.5 节）

图 114. ADC
连接示意图



注意

使用与 GPIO 引脚共享的 ADC 输入时，必须通过将该引脚的垫控寄存器中的IE 置低和OD 置高以禁用该引脚的数字功能。详见第2.19.6.3节，“焊盘控制 - 用户银行”。最大ADC输入电压由数字IO电源电压（IOVDD）决定，而非ADC电源电压（ADC_AVDD）。例如，若IOVDD供电为1.8V，即使ADC_AVDD供电为3.3V，ADC输入电压也不得超过1.8V。超过IOVDD的电压将通过ESD保护二极管产生泄漏电流。详见第5.5.3节，“引脚规格”。

4.9.1. ADC 控制器

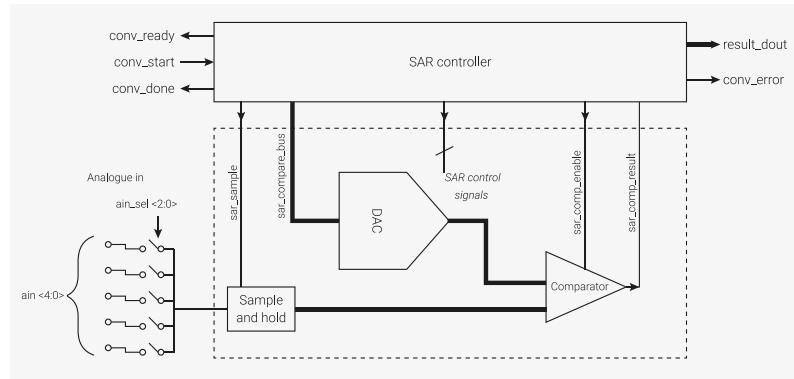
数字控制器负责管理RP2040 ADC的具体操作，并提供以下附加功能：

- 单次采集或自由运行采样模式
- 带DMA接口的采样FIFO
- 用于设定自由运行采样率的定时器（16位整数，8位小数）
- 自由运行采样模式下的多通道轮询采样
- 自由运行采样模式下，可选择右移至8位，以便通过DMA传输至系统内存中的字节缓冲区

4.9.2. SAR ADC

The SAR ADC (Successive Approximation Register Analogue to Digital Converter) is a combination of digital controller, and analogue circuit as shown in [Figure 115](#).

[Figure 115. SAR ADC Block diagram](#)



The ADC requires a 48MHz clock (`clk_adc`), which could come from the USB PLL. Capturing a sample takes 96 clock cycles ($96 \times 1/48\text{MHz} = 2\mu\text{s}$ per sample (500ksps). The clock must be set up correctly before enabling the ADC.

Once the ADC block is provided with a clock, and its reset has been removed, writing a 1 to `CS.EN` will start a short internal power-up sequence for the ADC's analogue hardware. After a few clock cycles, `CS.READY` will go high, indicating the ADC is ready to start its first conversion.

The ADC can be disabled again at any time by clearing `CS.EN`, to save power. `CS.EN` does **not** enable the temperature sensor bias source (see [Section 4.9.5](#)). This is controlled separately.

The ADC input is capacitive, and when sampling, it places about 1pF across the input (there will be additional capacitance from outside the ADC, such as packaging and PCB routing, to add to this). The effective impedance, even when sampling at 500ksps, is over 100kΩ, and for DC measurements there should be no need to buffer.

4.9.2.1. One-shot Sample

Writing a 1 to `CS.START_ONCE` will immediately start a new conversion. `CS.READY` will go low, to show that a conversion is currently in progress. After 96 cycles of `clk_adc`, `CS.READY` will go high. The 12-bit conversion result is available in `RESULT`.

The ADC input to be sampled is selected by writing to `CS.AINSEL`, any time before the conversion starts. An `AINSEL` value of 0...3 selects the ADC input on GPIO 26...29. `AINSEL` of 4 selects the internal temperature sensor.

i NOTE

No settling time is required when switching `AINSEL`.

4.9.2.2. Free-running Sampling

When `CS.START_MANY` is set, the ADC will automatically start new conversions at regular intervals. The most recent conversion result is always available in `RESULT`, but for IRQ or DMA driven streaming of samples, the ADC FIFO must be enabled ([Section 4.9.2.4](#)).

By default (`DIV` = 0), new conversions start immediately upon the previous conversion finishing, so a new sample is produced every 96 cycles. At a clock frequency of 48MHz, this produces 500ksps.

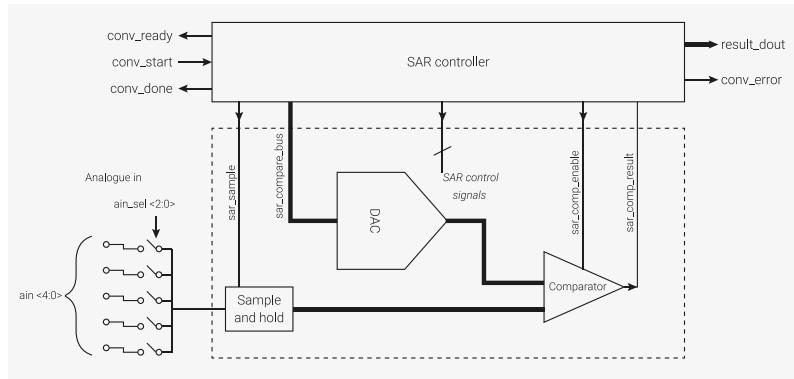
Setting `DIV.INT` to some positive value n will trigger the ADC once per $n + 1$ cycles, though the ADC ignores this if a conversion is currently in progress, so generally n will be ≥ 96 . For example, setting `DIV.INT` to 47999 will run the ADC at 1ksps, if running from a 48MHz clock.

The pacing timer supports fractional-rate division (first order delta sigma). When setting `DIV.FRAC` to a nonzero value,

4.9.2. SAR型ADC

SAR ADC（逐次逼近寄存器模数转换器）是数字控制器与模拟电路的组合，如图115所示。

图115。SAR ADC
框图



ADC需要48MHz时钟（`clk_adc`），该时钟可由USB PLL提供。采样耗时为96个时钟周期 ($96 \times 1/48\text{MHz} = 2\mu\text{s}$ 每样本 (500ksps))。启用ADC之前，时钟必须正确配置。

一旦ADC模块获得时钟且复位解除，向CS.EN写入1将启动ADC模拟硬件的短暂内部上电序列。经过数个时钟周期后，CS.READY信号将置高，表示ADC已准备开始首次转换。

可通过清除CS.EN随时禁用ADC，以节省功耗。CS.EN不启用温度传感器偏置电源（见第4.9.5节），此项由独立控制。

ADC输入为电容性，采样时会在输入端施加约1pF电容（ADC外部如封装和PCB走线还会增加额外电容）。即使以500ksps采样，有效阻抗仍高于100kΩ，对于直流测量通常无需缓冲。

4.9.2.1 单次采样

向CS.START_ONCE写入1将立即启动新的一次转换。CS.READY将置低，表示转换正在进行中。经过96个`clk_adc`周期后，CS.READY将置高。12位转换结果可在RESULT中获得。

采样的ADC输入可通过在转换开始前任意时刻写入CS.AINSEL进行选择。当AINSEL值为0~3时，选择GPIO 26~29作为ADC输入。当AINSEL值为4时，选择内部温度传感器。

注意

切换AINSEL时无需等待稳定时间。

4.9.2.2. 自由运行采样

当CS.START_MANY置位时，ADC将自动以固定间隔启动新的转换。最新的转换结果始终可在RESULT中读取，但对于通过IRQ或DMA进行的采样流，必须启用ADC FIFO（参见第4.9.2.4节）。

默认情况下(DIV=0)，新转换在前一次转换完成后立即开始，因此每96个时钟周期产生一个新样本。在48MHz时钟频率下，相当于500ksps。

将DIV.INT设置为某个正值n会使ADC每n+1个周期触发一次转换，但若当前转换尚未完成，ADC将忽略该设置，因此n通常需要满足 ≥ 96 。例如，在48MHz时钟频率下，将DIV.INT设置为47999，ADC可按1ksps运行。

定时器支持分数倍率分频（一阶三角Σ-Δ调制）。当将DIV.FRAC设置为非零值时，

the ADC will start a new conversion once per $1 + \text{INT} + \frac{\text{FRAC}}{256}$ cycles on average, by changing the sample interval between INT + 1 and INT + 2.

4.9.2.3. Sampling Multiple Inputs

[CS.RROBIN](#) allows the ADC to sample multiple inputs, in an interleaved fashion, while performing free-running sampling. Each bit in RROBIN corresponds to one of the five possible values of [CS.AINSEL](#). When the ADC completes a conversion, [CS.AINSEL](#) will automatically cycle to the next input whose corresponding bit is set in RROBIN.

The round-robin sampling feature is disabled by writing all-zeroes to [CS.RROBIN](#).

For example, if AINSEL is initially 0, and RROBIN is set to 0x06 (bits 1 and 2 are set), the ADC will sample channels in the following order:

1. Channel 0
2. Channel 1
3. Channel 2
4. Channel 1
5. Channel 2
6. Channel 1...

NOTE

The initial value of AINSEL does not need to correspond with a set bit in RROBIN.

4.9.2.4. Sample FIFO

The ADC samples can be read directly from the [RESULT](#) register, or stored in a local 8-entry FIFO and read out from [FIFO](#). FIFO operation is controlled by the [FCS](#) register.

If [FCS.EN](#) is set, the result of each ADC conversion is written to the FIFO. A software interrupt handler or the RP2040 DMA can read this sample from the FIFO when notified by the ADC's IRQ or DREQ signals. Alternatively, software can poll the status bits in [FCS](#) to wait for each sample to become available.

If the FIFO is full when a conversion completes, the sticky error flag [FCS.OVER](#) is set. The current FIFO contents are not changed by this event, but any conversion that completes whilst the FIFO is full will be lost.

There are two flags that control the data written to the FIFO by the ADC:

- [FCS.SHIFT](#) will right-shift the FIFO data to eight bits in size (i.e. FIFO bits 7:0 are conversion result bits 11:4). This is suitable for 8-bit DMA transfer to a byte buffer in memory, allowing deeper capture buffers, at the cost of some precision.
- [FCS.ERR](#) will set the [FIFO.ERR](#) flag of each FIFO value, showing that a conversion error took place, i.e. the SAR failed to converge (see below)

ADC 将以平均每
介于 INT + 1 和 INT + 2.

$$1 + \text{INT} + \frac{\text{FRAC}}{256}$$

个周期启动一次新的转换，通过更改采样间隔

4.9.2.3. 多输入采样

[CS.RROBIN 允许 ADC 在自由运行采样过程中以轮询方式采样多个输入。](#)

RROBIN 寄存器中的每一位对应于 CS.AINSEL 的五个可能值之一。当 ADC 完成转换后，CS.AINSEL 会自动切换至 RROBIN 中对应位被设置的下一个输入。

将全零写入 CS.RROBIN 即可禁用轮询采样功能。

例如，若AINSEL 初始设为 0，且RROBIN 设置为 0x06（第 1 和第 2 位被设置），ADC 会按以下顺序采样通道：

1. 通道 0
2. 通道 1
3. 通道 2
4. 通道 1
5. 通道 2
6. 通道 1...

i 注意

AINSEL 的初始值无需与 RROBIN 中被置位的位相对应。

4.9.2.4. 采样 FIFO

ADC 采样值可以直接从 RESULT 寄存器读取，或存储在本地包含 8 个条目的 FIFO 中，再从 FIFO 中读取。FIFO 的操作由 FCS 寄存器控制。

如果设置了 FCS.EN，每次 ADC 转换结果均写入 FIFO。当 ADC 的 IRQ 或 DREQ 信号发出通知时，软件中断处理程序或 RP2040 DMA 可从 FIFO 中读取该采样。另外，软件也可通过轮询 FCS 的状态位以等待每个采样数据的可用。

若转换完成时 FIFO 已满，将设置粘滞错误标志 FCS.OVER。此事件不会更改当前 FIFO 内容，但所有在 FIFO 已满时完成的转换采样将被丢失。

有两个标志控制ADC向FIFO写入的数据：

- [FCS.SHIFT 会将 FIFO 数据右移至八位大小（即 FIFO 位 7:0 对应转换结果的位 11:4）](#)。此举适用于对内存中的字节缓冲区进行 8 位 DMA 传输，允许更深的捕获缓冲区，但会以牺牲部分精度为代价。
- [FCS.ERR 会设置每个 FIFO 值的 FIFO.ERR 标志，表示发生了转换错误，即 SAR 未能收敛（详见下文）。](#)

⚠ CAUTION

Conversion errors produce undefined results, and the corresponding sample should be discarded. They indicate that the comparison of one or more bits failed to complete in the time allowed. Normally this is caused by comparator metastability, i.e. the closer to the comparator threshold the input signal is, the longer it will take to make a decision. The high gain of the comparator reduces the probability that no decision is made.

4.9.2.5. DMA

The RP2040 DMA ([Section 2.5](#)) can fetch ADC samples from the sample FIFO, by performing a normal memory-mapped read on the [FIFO](#) register, paced by the [ADC_DREQ](#) system data request signal. The following must be considered:

- The sample FIFO must be enabled ([FCS.EN](#)) so that samples are written to it; the FIFO is disabled by default so that it does not inadvertently fill when the ADC is used for one-shot conversions.
- The ADC's data request handshake (DREQ) must be enabled, via [FCS.DREQ_EN](#).
- The DMA channel used for the transfer must select the [DREQ_ADC](#) data request signal ([Section 2.5.3.1](#)).
- The threshold for DREQ assertion ([FCS.THRESH](#)) should be set to 1, so that the DMA transfers as soon as a single sample is present in the FIFO. Note this is also the threshold used for IRQ assertion, so non-DMA use cases might prefer a higher value for less frequent interrupts.
- If the DMA transfer size is set to 8 bits, so that the DMA transfers to a byte array in memory, [FCS.SHIFT](#) must also be set, to pre-shift the FIFO samples to 8 bits of significance.
- If multiple input channels are to be sampled, [CS.RROBIN](#) contains a 5-bit mask of those channels (4 external inputs plus temperature sensor). Additionally [CS.AINSEL](#) must select the channel for the first sample.
- The ADC sample rate ([Section 4.9.2.2](#)) should be configured before starting the ADC.

Once the ADC is suitably configured, the DMA channel should be started first, and the ADC conversion should be started second, via [CS.START_MANY](#). Once the DMA completes, the ADC can be halted, or a new DMA transfer promptly started. After clearing [CS.START_MANY](#) to halt the ADC, software should also poll [CS.READY](#) to make sure the last conversion has finished, and then drain any stray samples from the FIFO.

4.9.2.6. Interrupts

An interrupt can be generated when the FIFO level reaches a configurable threshold [FCS.THRESH](#). The interrupt output must be enabled via [INTE](#).

Status can be read from [INTS](#). The interrupt is cleared by draining the FIFO to a level lower than [FCS.THRESH](#).

4.9.2.7. Supply

The ADC supply is separated out on its own pin to allow noise filtering.

4.9.3. ADC ENOB

The ADC was characterised and the ENOB of the ADC was measured. Testing was carried out at room temperature across silicon lots, with tests being done on 3 typical ([tt](#)) as well as 3 fast ([ff](#)) and 3 slow ([ss](#)) corner RP2040 devices. The typical, minimum, and maximum values in [Table 566](#) reflect the silicon used in the testing.

Table 565. Parameters used during the testing.

Parameter	Value
Sample rate	250ksps

⚠ 注意

转换错误会产生未定义结果，应丢弃相应的采样数据。这些标志表明一个或多个位的比较未能在允许时间内完成。通常这是由比较器亚稳态引起，即输入信号越接近比较器阈值，作出决策所需时间越长。

比较器的高增益降低了未能作出决策的概率。

4.9.2.5. DMA

RP2040 DMA（第2.5节）可通过对FIFO寄存器进行普通内存映射读取，从样本FIFO获取ADC采样，该读取由 [ADC_DREQ](#) 系统数据请求信号控制。须考虑以下事项：

- 必须启用样本FIFO（FCS.EN），以允许样本写入其中；FIFO默认处于禁用状态，以防止ADC用于单次转换时 FIFO被意外填满。
- 必须通过FCS.DREQ_EN启用ADC数据请求握手（DREQ）。
- 用于传输的DMA通道必须选择 [DREQ_ADC](#) 数据请求信号（第2.5.3.1节）。
- DREQ断言阈值（FCS.THRESH）应设置为1，以确保FIFO中只要存在单个样本，DMA即刻开始传输。请注意，该阈值也是IRQ断言所用，非DMA应用场景可能会优先选择更高阈值以减少中断频率。
- 如果DMA传输大小设置为8位，即DMA传输至内存中的字节数组，则FCS.SHIFT必须设置，以预先将FIFO采样数据左移至8位有效位。
- 若需采样多个输入通道，CS.RROBIN包含一个5位掩码，标识这些通道（4个外部输入及温度传感器）。此外，CS.A_INSEL须选择首次采样的通道。
- 应在启动ADC之前配置ADC采样率（见第4.9.2.2节）。

ADC配置完成后，应先启动DMA通道，再通过CS.START_MANY启动ADC转换。DMA完成后，可停止ADC，或立即开始新的DMA传输。清除CS.START_MANY以停止ADC后，软件应轮询CS.READY以确认最后转换完成，并清空FIFO中残留的样本。

4.9.2.6. 中断

当 FIFO 水位达到可配置阈值 FCS.THRESH 时，可触发中断。中断输出须通过 INTEN 使能。

状态可通过 INTS 读取。中断通过将 FIFO 排空至低于 FCS.THRESH 水位来清除。

4.9.2.7. 电源

ADC 电源被独立分接至专用引脚以便进行噪声滤波。

4.9.3. ADC 有效位数 (ENOB)

对 ADC 进行了性能表征，并测量了 ADC 的 ENOB。测试在室温下对不同硅片批次实施，涵盖 3 个典型 ([tt](#))、3 个快速 ([ff](#)) 及 3 个慢速 ([ss](#)) 工艺边界 RP2040 器件。

表 566 中的典型值、最小值和最大值反映了用于测试的硅片特性。

表 565。 测试中使用的参数。

参数	数值
采样率	250ksps

Parameter	Value
FFT window	5 term Blackman-Harris
FFT bins	4,096
FFT averaging	none
Input level min	1
Input level max	4,094
Input frequency	997Hz

It should be noted that THD is normally calculated using the first 5 or 6 harmonics. However as INL/DNL errors (see [Section 4.9.4](#)) create more than this, the first 30 peaks are used. This makes the THD value slightly worse, but more representative of reality.

Table 566. Results for various parts tested (fast, slow, and typical).

	Min	Typical	Max
THD ¹	-55.6dB	55dB	-54.4dB
SNR	60.9dB	61.5dB	62.0dB
SFDR	59.2dB	59.9dB	60.5dB
SINAD	53.6dB	54.0dB	54.6dB
ENOB	8.6	8.7	8.8

¹ As the INL creates a large number of harmonics, the highest 30 peaks were used. This is different from conventional calculations of THD.

❗️ IMPORTANT

Testing was carried out using a board with a low-noise on-board voltage reference as, when characterising the ADC, it is important that there are no other noise sources affecting the measurements.

4.9.4. INL and DNL

Integral Non-Linearity (INL) and Differential Non-Linearity (DNL) are used to measure the error of the quantisation of the incoming signal that the ADC generates. In an ideal ADC the input-to-output transfer function should have a linear quantised transfer between the analogue input signal and the digitised output signal. The RP2040 ADC INL values for each binary result are shown in [Figure 116](#), illustrating that the error is a sawtooth rather than the expected curve.

参数	数值
FFT 窗函数	5 项 Blackman-Harris 窗
FFT 频段	4,096
FFT 平均值	无
输入电平最小值	1
输入电平最大值	4,094
输入频率	997Hz

应注意，THD通常使用前5或6个谐波进行计算。然而，由于INL/DNL误差（参见第4.9.4节）产生的峰值多于此数量，故采用前30个峰值。这导致THD值略微偏高，但更能反映实际情况。

表566。不同部件的测试结果（快速、慢速及典型）。

	最小值	典型值	最大值
THD ¹	-55.6dB	55dB	-54.4dB
信噪比	60.9dB	61.5dB	62.0dB
SFDR	59.2dB	59.9dB	60.5dB
SINAD	53.6dB	54.0dB	54.6dB
ENOB	8.6	8.7	8.8

¹由于INL产生大量谐波，因此采用了最高的30个峰值。这与传统的THD计算方法不同。

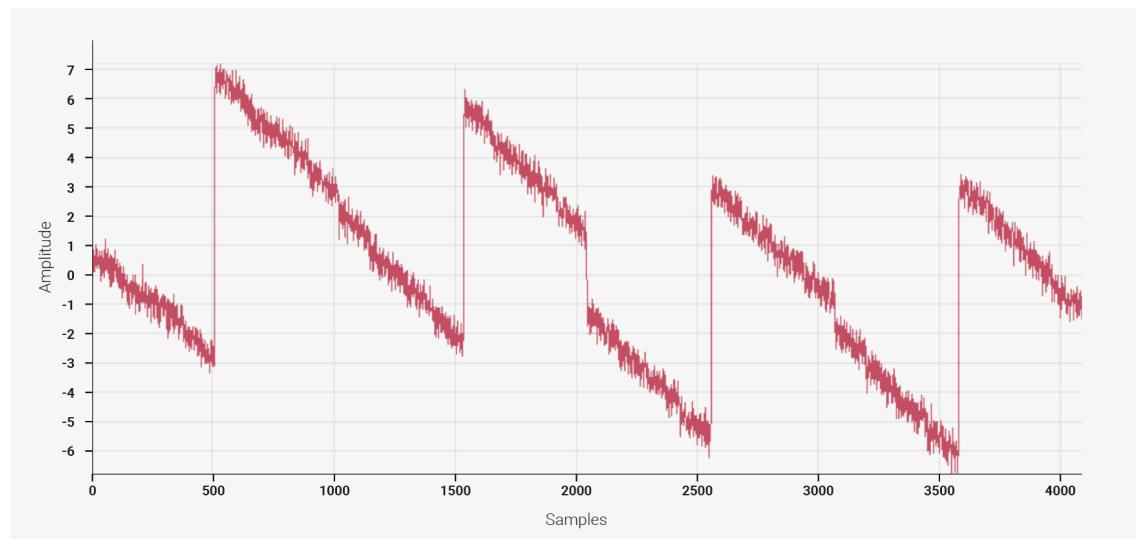
！重要

测试采用带有低噪声板载电压基准的板卡进行，因为在对ADC进行特性测试时，确保无其他噪声源影响测量至关重要。

4.9.4. 积分非线性 (INL) 与差分非线性 (DNL)

积分非线性 (INL) 与微分非线性 (DNL) 用于衡量ADC对输入信号量化误差的大小。理想 ADC 中，输入到输出的传输函数应在线性量化传递模拟输入信号与数字输出信号之间。RP2040 ADC中每个二进制结果对应的INL值如图116所示，表明误差呈锯齿形而非预期曲线。

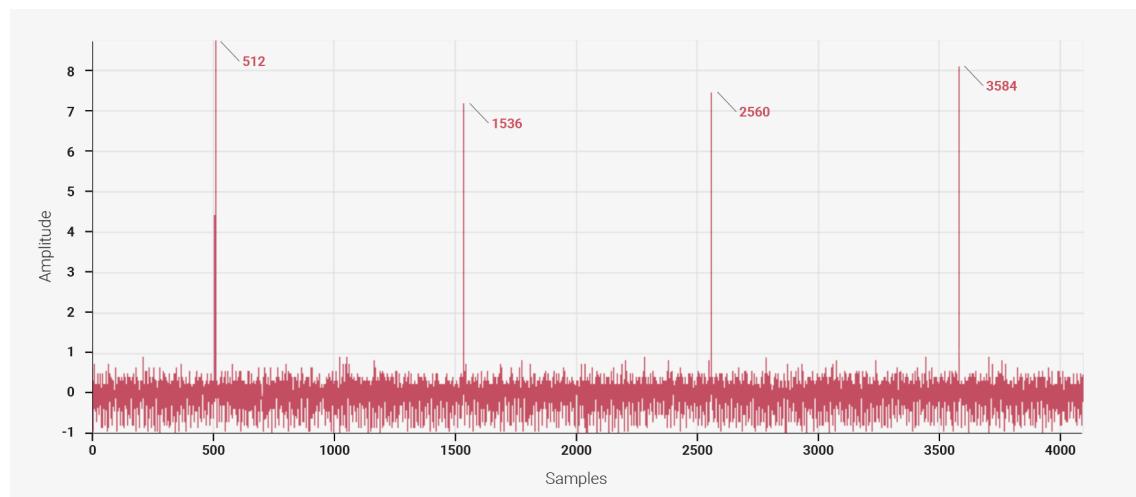
Figure 116. ATE machine results for INL (RP2040).



Nominally an ADC moves from one digital value to the next digital value, colloquially expressed as “no missing codes”. However, if the ADC skips a value bin this would cause a spike in the Differential Non-Linearity (DNL) error. These types of error often only occur at specific codes due to the design of the ADC.

The RP2040 ADC has a DNL which is mostly flat, and below 1 LSB. However at four values – 512, 1,536, 2,560, and 3,584 – the ADC’s DNL error peaks, see [Figure 117](#)

Figure 117. ATE machine results for DNL (RP2040).



The INL and DNL errors come from an error in the scaling of some internal capacitors of the ADC. These capacitors are small in value (only tens of femto Farads) and at these very small values, chip simulation of these capacitors can deviate slightly from reality. If these capacitors had matched correctly, the ADCs performance could have been better.

These INL and DNL errors will somewhat limit the performance of the ADC dependent on use case (See Errata [RP2040-E11](#)).

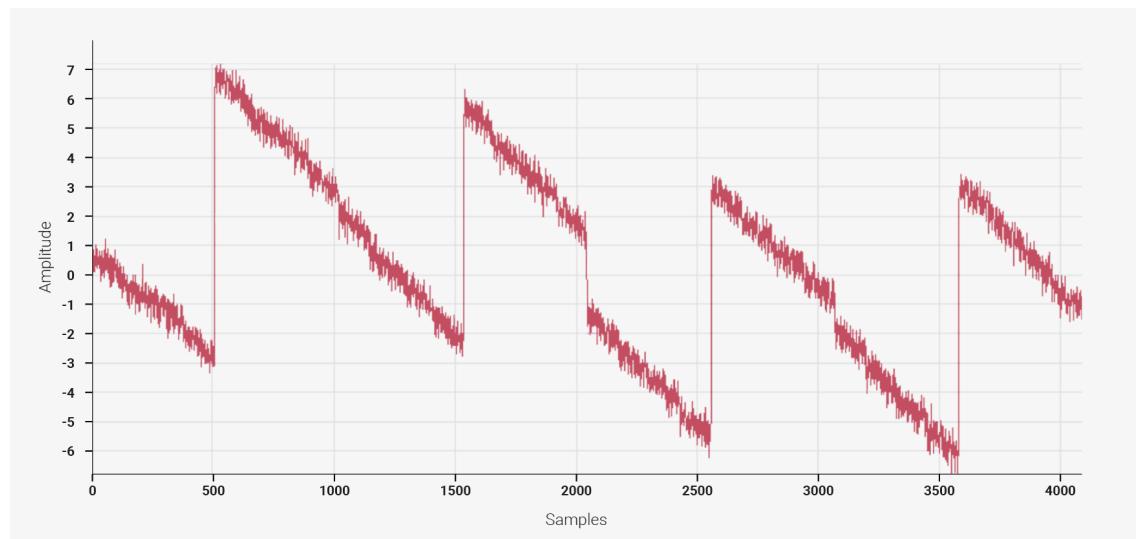
4.9.5. Temperature Sensor

The temperature sensor measures the V_{be} voltage of a biased bipolar diode, connected to the fifth ADC channel (AINSEL=4). Typically, V_{be} = 0.706V at 27 degrees C, with a slope of -1.721mV per degree. Therefore the temperature can be approximated as follows:

$$T = 27 - (\text{ADC_voltage} - 0.706)/0.001721$$

As the V_{be} and the V_{be} slope can vary over the temperature range, and from device to device, some user calibration may be required if accurate measurements are required.

图116。ATE
设备测试结果
INL (RP2040)。

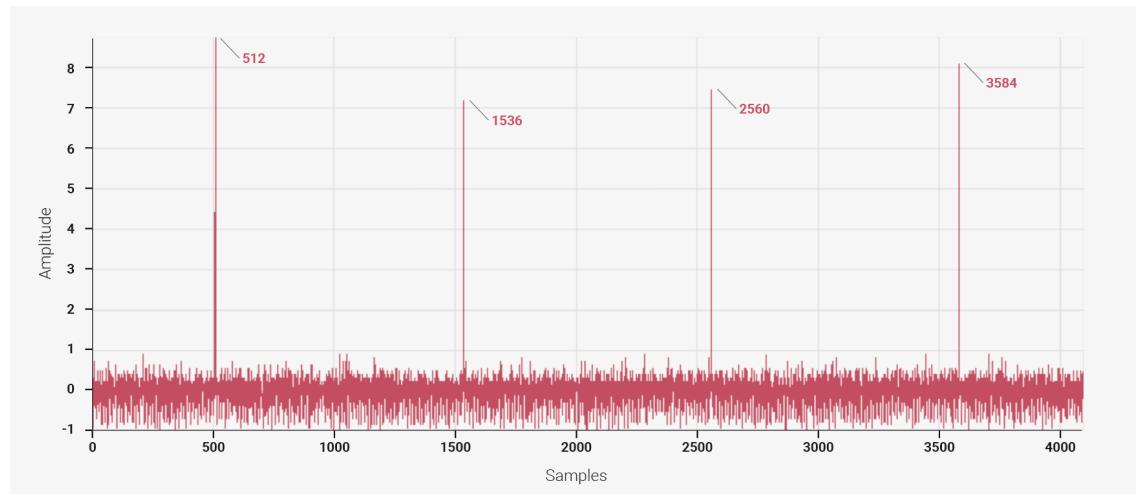


名义上，ADC从一个数字值移动到下一个数字值，通俗地称为“无缺码”。

然而，若ADC跳过某个数值区间，将导致差分非线性（DNL）误差出现尖峰。此类误差通常仅在特定码值处出现，系由ADC设计所致。

RP2040的ADC具有基本平坦且低于1 LSB的DNL，但在512、1536、2560及3584四个数值处，其DNL误差达到峰值，见图117。

图117。ATE
设备测试结果
DNL (RP2040)。



INL与DNL误差源于ADC内部部分电容的缩放比例偏差。因这些电容数值极小（仅数十飞法拉），芯片仿真结果在此微小数值段可能与实际有细微偏差。如果这些电容器匹配正确，ADC的性能本可更优。

这些INL和DNL误差将依据具体使用情况，在一定程度上限制ADC的性能（参见勘误 RP2040-E11）。

4.9.5. 温度传感器

温度传感器测量一个偏置的双极性二极管的V_{be}电压，该二极管连接至第五ADC通道（AINSEL=4）。通常，V_{be}在27摄氏度时为0.706V，斜率为每摄氏度-1.721mV。因此，温度可近似计算如下：

$$T = 27 - (\text{ADC_voltage} - 0.706)/0.001721$$

鉴于V_{be}及其斜率会随温度及器件差异变化，若需精确测量，用户可能需要进行校准。

The temperature sensor's bias source must be enabled before use, via `CS.TS_EN`. This increases current consumption on ADC_AVDD by approximately 40 μ A.

NOTE

The on board temperature sensor is very sensitive to errors in the reference voltage. If the ADC returns a value of 891 this would correspond to a temperature of 20.1°C. However if the reference voltage is 1% lower than 3.3V then the same reading of 891 would correspond to 24.3°C. You would see a change in temperature of over 4°C for a small 1% change in reference voltage. Therefore if you want to improve the accuracy of the internal temperature sensor it is worth considering adding an external reference voltage.

NOTE

The INL errors, see [Section 4.9.4](#), aren't in the usable temperature range of the ADC.

4.9.6. List of Registers

The ADC registers start at a base address of `0x4004c000` (defined as `ADC_BASE` in SDK).

Table 567. List of ADC registers

Offset	Name	Info
0x00	<code>CS</code>	ADC Control and Status
0x04	<code>RESULT</code>	Result of most recent ADC conversion
0x08	<code>FCS</code>	FIFO control and status
0x0c	<code>FIFO</code>	Conversion result FIFO
0x10	<code>DIV</code>	Clock divider. If non-zero, <code>CS_START_MANY</code> will start conversions at regular intervals rather than back-to-back. The divider is reset when either of these fields are written. Total period is $1 + \text{INT} + \text{FRAC} / 256$
0x14	<code>INTR</code>	Raw Interrupts
0x18	<code>INTE</code>	Interrupt Enable
0x1c	<code>INTF</code>	Interrupt Force
0x20	<code>INTS</code>	Interrupt status after masking & forcing

ADC: CS Register

Offset: 0x00

Description

ADC Control and Status

Table 568. CS Register

Bits	Description	Type	Reset
31:21	Reserved.	-	-
20:16	RROBIN: Round-robin sampling. 1 bit per channel. Set all bits to 0 to disable. Otherwise, the ADC will cycle through each enabled channel in a round-robin fashion. The first channel to be sampled will be the one currently indicated by <code>AINSEL</code> . <code>AINSEL</code> will be updated after each conversion with the newly-selected channel.	RW	0x00
15	Reserved.	-	-

温度传感器的偏置源必须通过 CS_TS_EN 使能后方可使用，此操作会使 ADC_AVDD 电流增加约 40 μ A。

① 注意

板载温度传感器对基准电压的误差极为敏感。若ADC返回值为891，则对应温度为20.1°C。然而，若参考电压比3.3V低1%，则相同的891读数对应的温度为24.3°C。参考电压仅1%的微小变化，即引起温度变化超过4°C。因此，若您希望提高内部温度传感器的精度，建议考虑添加外部参考电压。

① 注意

INL误差（参见第4.9.4节）不属于ADC的可用温度范围。

4.9.6. 寄存器列表

ADC寄存器的基址起始为 **0x4004c000**（在SDK中定义为ADC_BASE）。

表 567. ADC 寄存器列表

偏移量	名称	说明
0x00	CS	ADC控制与状态
0x04	RESULT	最近一次ADC转换的结果
0x08	FCS	FIFO控制与状态
0x0c	FIFO	转换结果 FIFO
0x10	DIV	时钟分频器。若非零，CS_START_MANY 会启动转换 以固定间隔而非连续进行。 任一字段写入时，分频器将被重置。 总周期为 $1 + INT + FRAC / 256$
0x14	INTR	原始中断
0x18	INTE	中断使能
0x1c	INTF	中断强制
0x20	INTS	掩码及强制后的中断状态

ADC：CS 寄存器

偏移: 0x00

描述

ADC控制与状态

表 568: CS 寄存器

位	描述	类型	复位值
31:21	保留。	-	-
20:16	RROBIN : 轮询采样，每通道 1 位。将所有位设置为 0 以禁用。 否则，ADC 将以轮询方式依次转换每个启用的通道。 首次采样通道为当前AINSEL指定的通道。 AINSEL会在每次转换后自动更新为新选通道。	读写	0x00
15	保留。	-	-

Bits	Description	Type	Reset
14:12	AINSEL : Select analog mux input. Updated automatically in round-robin mode.	RW	0x0
11	Reserved.	-	-
10	ERR_STICKY : Some past ADC conversion encountered an error. Write 1 to clear.	WC	0x0
9	ERR : The most recent ADC conversion encountered an error; result is undefined or noisy.	RO	0x0
8	READY : 1 if the ADC is ready to start a new conversion. Implies any previous conversion has completed. 0 whilst conversion in progress.	RO	0x0
7:4	Reserved.	-	-
3	START_MANY : Continuously perform conversions whilst this bit is 1. A new conversion will start immediately after the previous finishes.	RW	0x0
2	START_ONCE : Start a single conversion. Self-clearing. Ignored if start_many is asserted.	SC	0x0
1	TS_EN : Power on temperature sensor. 1 - enabled. 0 - disabled.	RW	0x0
0	EN : Power on ADC and enable its clock. 1 - enabled. 0 - disabled.	RW	0x0

ADC: RESULT Register

Offset: 0x04

Table 569. RESULT Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:0	Result of most recent ADC conversion	RO	0x000

ADC: FCS Register

Offset: 0x08

Description

FIFO control and status

Table 570. FCS Register

Bits	Description	Type	Reset
31:28	Reserved.	-	-
27:24	THRESH : DREQ/IRQ asserted when level \geq threshold	RW	0x0
23:20	Reserved.	-	-
19:16	LEVEL : The number of conversion results currently waiting in the FIFO	RO	0x0
15:12	Reserved.	-	-
11	OVER : 1 if the FIFO has been overflowed. Write 1 to clear.	WC	0x0
10	UNDER : 1 if the FIFO has been underflowed. Write 1 to clear.	WC	0x0
9	FULL	RO	0x0
8	EMPTY	RO	0x0
7:4	Reserved.	-	-

位	描述	类型	复位值
14:12	AINSEL : 选择模拟多路复用输入。在轮询模式下自动更新。	读写	0x0
11	保留。	-	-
10	ERR_STICKY : 先前某次 ADC 转换发生错误，写 1 清除。	WC	0x0
9	ERR : 最近一次 ADC 转换发生错误；结果未定义或含噪声。	只读	0x0
8	READY : 当ADC准备开始新转换时置1。表示先前的转换已完成。 转换进行中时置0。	只读	0x0
7:4	保留。	-	-
3	START_MANY : 当该位为1时，连续执行转换。前一次转换结束后立即开始新转换。	读写	0x0
2	START_ONCE : 启动单次转换。自动清零。若start_many置位则忽略此位。	SC	0x0
1	TS_EN : 温度传感器上电。1表示启用，0表示禁用。	读写	0x0
0	EN : ADC上电并使能时钟。 1表示启用，0表示禁用。	读写	0x0

ADC: RESULT寄存器

偏移: 0x04

表569. RESULT
寄存器

位	描述	类型	复位值
31:12	保留。	-	-
11:0	最近一次ADC转换的结果	只读	0x000

ADC: FCS寄存器

偏移: 0x08

描述

FIFO控制与状态

表570. FCS
寄存器

位	描述	类型	复位值
31:28	保留。	-	-
27:24	THRESH : 当电平 \geq 阈值时，DREQ/IRQ被触发	读写	0x0
23:20	保留。	-	-
19:16	LEVEL : FIFO中当前等待的转换结果数量	只读	0x0
15:12	保留。	-	-
11	OVER : FIFO溢出时置1。写1以清除该状态。	WC	0x0
10	UNDER : FIFO欠流时置1。写1以清除该状态。	WC	0x0
9	满	只读	0x0
8	空	只读	0x0
7:4	保留。	-	-

Bits	Description	Type	Reset
3	DREQ_EN: If 1: assert DMA requests when FIFO contains data	RW	0x0
2	ERR: If 1: conversion error bit appears in the FIFO alongside the result	RW	0x0
1	SHIFT: If 1: FIFO results are right-shifted to be one byte in size. Enables DMA to byte buffers.	RW	0x0
0	EN: If 1: write result to the FIFO after each conversion.	RW	0x0

ADC: FIFO Register

Offset: 0x0c

Description

Conversion result FIFO

Table 571. FIFO Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15	ERR: 1 if this particular sample experienced a conversion error. Remains in the same location if the sample is shifted.	RF	-
14:12	Reserved.	-	-
11:0	VAL	RF	-

ADC: DIV Register

Offset: 0x10

Description

Clock divider. If non-zero, CS_START_MANY will start conversions at regular intervals rather than back-to-back.

The divider is reset when either of these fields are written.

Total period is $1 + \text{INT} + \text{FRAC} / 256$

Table 572. DIV Register

Bits	Description	Type	Reset
31:24	Reserved.	-	-
23:8	INT: Integer part of clock divisor.	RW	0x0000
7:0	FRAC: Fractional part of clock divisor. First-order delta-sigma.	RW	0x00

ADC: INTR Register

Offset: 0x14

Description

Raw Interrupts

位	描述	类型	复位值
3	DREQ_EN : 若为1，当FIFO中含有数据时，断言DMA请求	读写	0x0
2	ERR : 若为1，转换错误位将与结果一同出现在FIFO中	读写	0x0
1	SHIFT : 若为1，FIFO结果向右移位，变为一字节大小。启用对字节缓冲区的DMA。	读写	0x0
0	EN : 若为1，每次转换后将结果写入FIFO。	读写	0x0

ADC: FIFO寄存器

偏移: 0x0c

描述

转换结果 FIFO

表571. FIFO
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15	ERR : 若此特定采样发生转换错误，则为1。若采样发生移位，则保持在相同位置。	RF	-
14:12	保留。	-	-
11:0	值	RF	-

ADC: DIV寄存器

偏移: 0x10

说明

时钟分频器。若非零，CS_START_MANY将按固定间隔启动转换，而非连续启动。

任一字段写入时，分频器将被重置。

总周期为 $1 + \text{INT} + \text{FRAC} / 256$

表572. DIV
寄存器

位	描述	类型	复位值
31:24	保留。	-	-
23:8	INT : 时钟分频器的整数部分。	读写	0x0000
7:0	FRAC : 时钟分频器的小数部分。一级 Δ - Σ 调制。	读写	0x00

ADC: INTR寄存器

偏移: 0x14

描述

原始中断

Table 573. INTR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	FIFO: Triggered when the sample FIFO reaches a certain level. This level can be programmed via the FCS_THRESH field.	RO	0x0

ADC: INTE Register

Offset: 0x18

Description

Interrupt Enable

Table 574. INTE Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	FIFO: Triggered when the sample FIFO reaches a certain level. This level can be programmed via the FCS_THRESH field.	RW	0x0

ADC: INTF Register

Offset: 0x1c

Description

Interrupt Force

Table 575. INTF Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	FIFO: Triggered when the sample FIFO reaches a certain level. This level can be programmed via the FCS_THRESH field.	RW	0x0

ADC: INTS Register

Offset: 0x20

Description

Interrupt status after masking & forcing

Table 576. INTS Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	FIFO: Triggered when the sample FIFO reaches a certain level. This level can be programmed via the FCS_THRESH field.	RO	0x0

4.10. SSI

Synopsys Documentation

Synopsys Proprietary. Used with permission.

RP2040 has a Synchronous Serial Interface (SSI) controller which appears on the QSPI pins and is used to communicate with external Flash devices. The SSI forms part of the XIP block.

The SSI controller is based on a configuration of the Synopsys DW_apb_ssi IP (v4.01a).

表 573. INTR 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	只读	0x0

ADC: INTE 寄存器

偏移: 0x18

描述

中断使能

表 574. INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	读写	0x0

ADC: INTF 寄存器

偏移: 0x1c

描述

中断强制

表 575. INTF 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	读写	0x0

ADC: INTS 寄存器

偏移: 0x20

说明

掩码及强制后的中断状态

表 576. INTS 寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	FIFO : 当采样 FIFO 达到某一设定水平时触发。 该水平可通过 FCS_THRESH 字段进行编程。	只读	0x0

4.10. 串行同步接口 (SSI)

Synopsys文档

Synopsys专有，经过授权使用。

RP2040 配备同步串行接口 (SSI) 控制器，该控制器位于 QSPI 引脚，用于与外部 Flash 设备通信。该 SSI 控制器属于 XIP 模块的一部分。

SSI 控制器基于 Synopsys DW_apb_ssi IP (版本 4.01a) 配置而成。

4.10.1. Overview

In order for the DW_apb_ssi to connect to a serial-master or serial-slave peripheral device, the peripheral must have at least one of the following interfaces:

Motorola Serial Peripheral Interface (SPI)

A four-wire, full-duplex serial protocol from Motorola. There are four possible combinations for the serial clock phase and polarity. The clock phase (SCPH) determines whether the serial transfer begins with the falling edge of the slave select signal or the first edge of the serial clock. The slave select line is held high when the DW_apb_ssi is idle or disabled.

Texas Instruments Serial Protocol (SSP)

A four-wire, full-duplex serial protocol. The slave select line used for SPI and Microwire protocols doubles as the frame indicator for the SSP protocol.

National Semiconductor Microwire

A half-duplex serial protocol, which uses a control word transmitted from the serial master to the target serial slave.

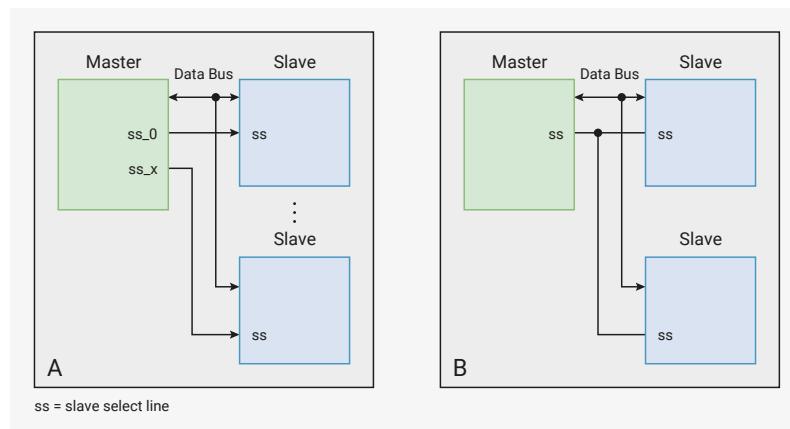
You can program the FRF (frame format) bit field in the Control Register 0 (CTRLR0) to select which protocol is used.

The serial protocols supported by the DW_apb_ssi allow for serial slaves to be selected or addressed using either hardware or software. When implemented in hardware, serial slaves are selected under the control of dedicated hardware select lines. The number of select lines generated from the serial master is equal to the number of serial slaves present on the bus. The serial-master device asserts the select line of the target serial slave before data transfer begins. This architecture is illustrated in [Figure 118](#).

When implemented in software, the input select line for all serial slave devices should originate from a single slave select output on the serial master. In this mode it is assumed that the serial master has only a single slave select output. If there are multiple serial masters in the system, the slave select output from all masters can be logically ANDed to generate a single slave select input for all serial slave devices. The main program in the software domain controls selection of the target slave device; this architecture is illustrated in [Figure 118](#). Software would use the SSIENR register in all slaves in order to control which slave is to respond to the serial transfer request from the master device.

The DW_apb_ssi does not enforce hardware or software control for serial-slave device selection. You can configure the DW_apb_ssi for either implementation, illustrated in [Figure 118](#).

Figure 118.
Hardware/Software
Slave Selection.



4.10.2. Features

The DW_apb_ssi is a configurable and programmable component that is a full-duplex master serial interface. The host processor accesses data, control, and status information on the DW_apb_ssi through the APB interface. The DW_apb_ssi also interfaces with the DMA Controller for bulk data transfer.

The DW_apb_ssi is configured as a serial master. The DW_apb_ssi can connect to any serial-slave peripheral device using one of the following interfaces:

4.10.1. 概述

为使 DW_apb_ssi 能连接至串行主设备或串行从设备外设，外设必须至少具备以下任一接口：

摩托罗拉串行外围接口（SPI）

摩托罗拉公司提出的一种四线全双工串行协议。串行时钟的相位及极性共有四种可能组合。时钟相位（SCPH）决定串行传输是从从设备选择信号的下降沿开始，还是从串行时钟的第一个边沿开始。当 DW_apb_ssi 处于空闲或禁用状态时，从设备选择线保持高电平。

德州仪器串行协议（SSP）

一种四线全双工串行协议。用于 SPI 和 Microwire 协议的从设备选择线兼作 SSP 协议的帧指示线。

国家半导体 Microwire 协议

一种半双工串行协议，通过串行主设备向目标串行从设备传输控制字。

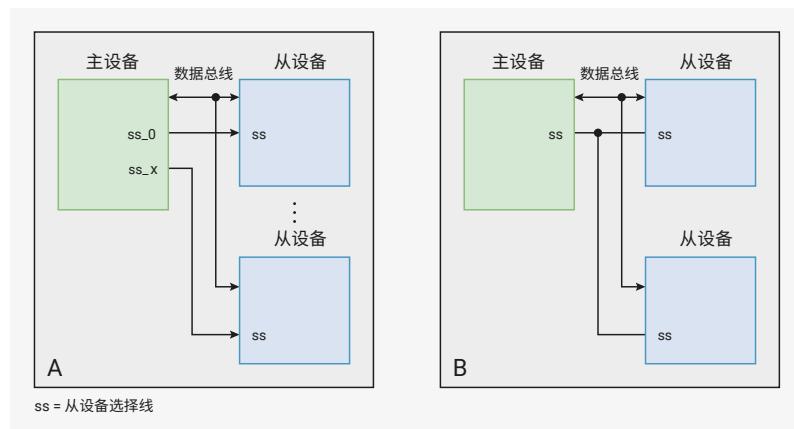
用户可通过控制寄存器 0 (CTRLR0) 中的 FRF (帧格式) 位字段编程，选择所用协议。

DW_apb_ssi 支持的串行协议允许使用硬件或软件方式选择或寻址串行从设备。在硬件实现时，串行从设备通过专用的硬件选择线进行选取。串行主设备产生的选择线数量应等于总线上的串行从设备数量。在数据传输开始之前，串行主设备会使目标串行从设备的选择线有效。该架构如图118所示。

在软件实现时，所有串行从设备的输入选择线应来自串行主设备的单一从设备选择输出。在此模式下，假设串行主设备仅有一个从设备选择输出。如果系统中存在多个串行主设备，则所有主设备的从设备选择输出可通过逻辑与操作生成所有串行从设备的单一从设备选择输入。软件主程序负责控制目标从设备的选择；该架构如图118所示。软件会利用所有从设备中的SSIENR寄存器来控制响应主设备串行传输请求的从设备。

DW_apb_ssi 不对串行从设备的硬件或软件控制进行强制管理。您可以依据图 118 所示配置 DW_apb_ssi，以实现任一方方案。

图 118。
硬件/软件
从设备选择。



4.10.2. 特性

DW_apb_ssi 是一款可配置且可编程的组件，作为全双工主序列接口。主处理器通过 APB 接口访问 DW_apb_ssi 的数据、控制及状态信息。DW_apb_ssi 还通过 DMA 控制器实现大批量数据传输。

DW_apb_ssi 配置为串行主设备。DW_apb_ssi 可通过以下任何接口连接至任意串行从外设设备：

- Motorola Serial Peripheral Interface (SPI)
- Texas Instruments Serial Protocol (SSP)
- National Semiconductor Microwire

On RP2040, the DW_apb_ssi is a component of the flash execute-in-place subsystem (see [Section 2.6.3](#)), and provides communication with an external SPI, dual-SPI or quad-SPI flash device.

4.10.2.1. IO connections

The SSI controller connects to the following pins:

- `QSPI_SCLK` Connected to output clock `sclk_out`
- `QSPI_SS_N` Connected to chip select `ss_o_n`
- `QSPI_SD[3:0]` Connected to data bus `txd` and `rxn`

Some pins on the IP are tied off as not used:

- `ss_in_n` is tied high

Clock connections are as follows:

- `pclk` and `sclk` are driven from `clk_sys`

4.10.3. IP Modifications

The following modifications were made to the Synopsys DW_apb_ssi hardware:

1. XIP accesses are byte-swapped, such that the least-addressed byte is in the least-significant position
2. When `SPI_CTRLR0_INST_L` is 0, the XIP instruction field is appended to the end of the address for XIP accesses, rather than prepended to the beginning
3. The reset value of `DMARDLR` is increased from 0 to 4. The SSI to DMA handshaking on RP2040 requests only single transfers or bursts of four, depending on whether the RX FIFO level has reached `DMARDLR`, so `DMARDLR` should not be changed from this value.

The first of these changes allows mixed-size accesses by a little-endian busmaster, such as the RP2040 DMA, or the Cortex-M0+ configuration used on RP2040. Note that this only applies to XIP accesses (RP2040 system addresses in the range `0x10000000` to `0x13fffff`), not to direct access to the DW_apb_ssi FIFOs. When accessing the SSI directly, it may be necessary for software to swap bytes manually, or to use the RP2040 DMA's byte swap feature.

The second supports issuing of continuation bits following the XIP address, so that command-prefix-free XIP modes can be supported (e.g. `EBh` Quad I/O Fast Read on Winbond devices), for greater performance. For example, the following configuration would be used to issue a standard `03h` serial read command for each access to the XIP address window:

- `SPI_CTRLR0_INST_L` = 8 bits
- `SPI_CTRLR0_ADDR_L` = 24 bits
- `SPI_CTRLR0_XIP_CMD` = `0x03`

This will first issue eight command bits (`0x03`), then issue 24 address bits, then clock in the data bits. The configuration used for `EBh` quad read, after the flash has entered the XIP state, would be:

- `SPI_CTRLR0_INST_L` = 0
- `SPI_CTRLR0_ADDR_L` = 32 bits
- `SPI_CTRLR0_XIP_CMD` = `0xa0` (continuation code on W25Qx devices)

For each XIP access, the DW_apb_ssi will issue 32 "address" bits, consisting of the 24 LSBs of the RP2040 system bus

- 摩托罗拉串行外围接口（SPI）
- 德州仪器串行协议（SSP）
- 国家半导体 Microwire 协议

在 RP2040 上，DW_apb_ssi 是闪存执行就地子系统的组成部分（参见第 2.6.3 节），提供与外部 SPI、双 SPI 或四 SPI 闪存设备的通信。

4.10.2.1. 输入/输出连接

SSI 控制器连接至以下引脚：

- `QSPI_SCLK` 连接至输出时钟 `sclk_out`
- `QSPI_SS_N` 连接至片选信号 `ss_o_n`
- `QSPI_SD[3:0]` 连接至数据总线 `txd` 及 `rxn`

IP 上部分引脚被固定为未使用状态：

- `ss_in_n` 被拉高

时钟连接如下：

- `pclk` 及 `sclk` 由 `clk_sys` 驱动

4.10.3. IP 修改

对 Synopsys DW_apb_ssi 硬件进行了以下修改：

1. XIP 访问采用字节交换方式，最低地址字节位于最低有效字节位置
2. 当 `SPI_CTRLR0_INST_L` 为 0 时，XIP 指令字段追加于地址末尾，而非前置于地址起始
3. 寄存器 `DMARDLR` 的复位值由 0 调整为 4。RP2040 上 SSI 与 DMA 的握手依据 RX FIFO 水平是否达到 `DMARDLR`，仅请求单次传输或四次突发传输，故 `DMARDLR` 不宜更改。

这些更改中的第一个允许小端总线主控设备进行混合大小访问，例如 RP2040 的 DMA 或 RP2040 上使用的 Cortex-M0+ 配置。请注意，此更改仅适用于 XIP 访问（RP2040 系统地址范围 `0x10000000` 至 `0x13fffff`），不适用于直接访问 DW_apb_ssi FIFO。直接访问 SSI 时，软件可能需手动进行字节交换，或使用 RP2040 DMA 的字节交换功能。

第二项更改支持在 XIP 地址后发出续传位，从而支持无命令前缀的 XIP 模式（例如 Winbond 设备的 `EBh` Quad I/O 快速读取），以提升性能。例如，以下配置用于对每次访问 XIP 地址窗口时发出标准的 `03h` 串行读取命令：

- `SPI_CTRLR0_INST_L = 8 bits`
- `SPI_CTRLR0_ADDR_L = 24 位`
- `SPI_CTRLR0_XIP_CMD = 0x03`

此操作将首先发出八位命令位（`0x03`），随后发出 24 位地址位，最后时钟输入数据位。用于 `EBh` 四线读取的配置，在闪存进入 XIP 状态后，配置如下：

- `SPI_CTRLR0_INST_L = 0`
- `SPI_CTRLR0_ADDR_L = 32 位`
- `SPI_CTRLR0_XIP_CMD = 0xa0` （适用于 W25Qx 设备的续传码）

每次 XIP 访问时，DW_apb_ssi 将发送 32 个“地址”位，其中包含 RP2040 系统总线最低 24 位地址。

address, followed by the 8-bit continuation code `0xa0`. No command prefix is issued.

4.10.3.1. Example of Target Slave Selection Using Software

The following example is pseudo code that illustrates how to use software to select the target slave.

```
1 int main() {
2     disable_all_serial_devices(); ①
3     initialize_mst(ssi_mst_1); ②
4     initialize_slv(ssi_slv_1); ③
5     start_serial_xfer(ssi_mst_1); ④
6 }
```

- | | | | |
|---|--|---|---|
| ① This function sets the SSI_EN bit to logic '0' in the SSIENR register of each device on the serial bus. | ② This function initializes the master device for the serial transfer; | ③ This function initializes the target slave device (slave 1 in this example) for the serial transfer; | ④ This function begins the serial transfer by writing transmit data into the master's TX FIFO. User can poll the busy status with a function or use an ISR to determine when the serial transfer has completed. |
| | <ol style="list-style-type: none"> 1. Write CTRLR0 to match the required transfer 2. If transfer is receive only write number of frames into CTRLR1 3. Write BAUDR to set the transfer baud rate. 4. Write TXFTLR and RXFTLR to set FIFO threshold levels 5. Write IMR register to set interrupt masks 6. Write SER register bit[0] to logic '1' 7. Write SSIENR register bit[0] to logic '1' to enable the master. | <ol style="list-style-type: none"> 1. Write CTRLR0 to match the required transfer 2. Write TXFTLR and RXFTLR to set FIFO threshold levels 3. Write IMR register to set interrupt masks 4. Write SSIENR register bit[0] to logic '1' to enable the slave. 5. If the slave is to transmit data, write data into TX FIFO Now the slave is enabled and awaiting an active level on its ss_in_n input port. Note all other serial slaves are disabled (SSI_EN=0) and therefore will not respond to an active level on their ss_in_n port. | |

4.10.4. Clock Ratios

The maximum frequency of the bit-rate clock (sclk_out) is one-half the frequency of ssi_clk. This allows the shift control logic to capture data on one clock edge of sclk_out and propagate data on the opposite edge.

[Figure 119](#) illustrates the maximum ratio between sclk_out and ssi_clk.

， 并紧随其后发送8位续传码 **0xa0**。 不发送命令前缀。

4.10.3.1. 使用软件进行目标从设备选择的示例

以下示例为伪代码，演示如何使用软件选择目标从设备。

```
1 int main() {
2     disable_all_serial_devices(); ①
3     initialize_mst(ssi_mst_1); ②
4     initialize_slv(ssi_slv_1); ③
5     start_serial_xfer(ssi_mst_1); ④
6 }
```

①此函数将串行总线中每个设备的SSIENR寄存器的SSI_EN位设置为逻辑'0'。

②此函数用于初始化串行传输的主设备；

③此函数初始化目标从设备（本例为从设备1），以进行串行传输；

④此函数通过将传输数据写入主设备的TX FIFO来启动串行传输。用户可通过函数轮询忙碌状态，或使用中断服务程序（ISR）判断串行传输何时完成。

1. Write CTRLR0 to match the required transfer
2. If transfer is receive only write number of frames into CTRLR1
3. Write BAUDR to set the transfer baud rate.
4. Write TXFTLR and RXFTLR to set FIFO threshold levels
5. Write IMR register to set interrupt masks
6. Write SER register bit[0] to logic '1'
7. Write SSIENR register bit[0] to logic '1' to enable the master.

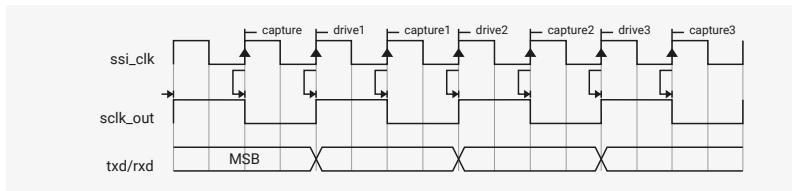
1. Write CTRLR0 to match the required transfer
2. Write TXFTLR and RXFTLR to set FIFO threshold levels
3. Write IMR register to set interrupt masks
4. 将SSIENR寄存器的第一位写为逻辑“1”，以启用从设备。
5. 若从设备需传输数据，则将数据写入TX FIFO。此时从设备已启用，正等待其ss_in_n输入端口的有效电平。请注意，所有其他串行从设备均被禁用（SSI_EN=0），因此不会响应其ss_in_n端口的有效电平。

4.10.4. 时钟比率

位速率时钟（sclk_out）的最大频率为ssi_clk频率的一半。该设计允许移位控制逻辑在sclk_out的一个时钟边沿采集数据，并在相反边沿传输数据。

[图119示意了sclk_out与ssi_clk之间的最大频率比。](#)

Figure 119. Maximum sclk_out/ssi_clk Ratio.



The sclk_out line toggles only when an active transfer is in progress. At all other times it is held in an inactive state, as defined by the serial protocol under which it operates.

The frequency of sclk_out can be derived from the following equation:

$$F_{sclk_out} = \frac{F_{ssi_clk}}{SCKDV}$$

SCKDV is a bit field in the programmable register BAUDR, holding any even value in the range 0 to 65,534. If SCKDV is 0, then sclk_out is disabled.

4.10.4.1. Frequency Ratio Summary

A summary of the frequency ratio restrictions between the bit-rate clock (sclk_out) and the DW_apb_ssi peripheral clock (ssi_clk) are as follows:

- $F_{ssi_clk} >= 2 \times (\text{maximum } F_{sclk_out})$

4.10.5. Transmit and Receive FIFO Buffers

The FIFO buffers used by the DW_apb_ssi are internal D-type flip-flops that are 16 entries deep. The width of both transmit and receive FIFO buffers is fixed at 32 bits, due to the serial specifications, which state that a serial transfer (data frame) can be 4 to 16/32 bits in length. Data frames that are less than 32 bits must be right-justified when written into the transmit FIFO buffer. The shift control logic automatically right-justifies receive data in the receive FIFO buffer.

Each data entry in the FIFO buffers contains a single data frame. It is impossible to store multiple data frames in a single FIFO location; for example, you may not store two 8-bit data frames in a single FIFO location. If an 8-bit data frame is required, the upper bits of the FIFO entry are ignored or unused when the serial shifter transmits the data.

NOTE

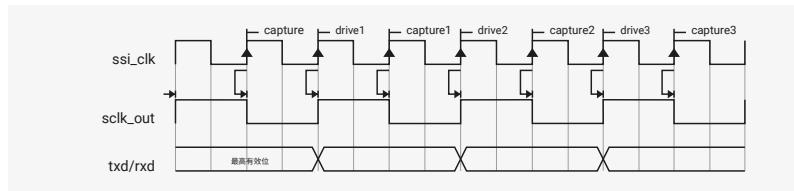
The transmit and receive FIFO buffers are cleared when the DW_apb_ssi is disabled (SSI_EN = 0) or when it is reset (preseth).

The transmit FIFO is loaded by APB write commands to the DW_apb_ssi data register (DR). Data are popped (removed) from the transmit FIFO by the shift control logic into the transmit shift register. The transmit FIFO generates a FIFO empty interrupt request (ssi_txe_intr) when the number of entries in the FIFO is less than or equal to the FIFO threshold value. The threshold value, set through the programmable register TXFTLR, determines the level of FIFO entries at which an interrupt is generated. The threshold value allows you to provide early indication to the processor that the transmit FIFO is nearly empty. A transmit FIFO overflow interrupt (ssi_txo_intr) is generated if you attempt to write data into an already full transmit FIFO.

Data are popped from the receive FIFO by APB read commands to the DW_apb_ssi data register (DR). The receive FIFO is loaded from the receive shift register by the shift control logic. The receive FIFO generates a FIFO-full interrupt request (ssi_rxf_intr) when the number of entries in the FIFO is greater than or equal to the FIFO threshold value plus one. The threshold value, set through programmable register RXFTLR, determines the level of FIFO entries at which an interrupt is generated.

The threshold value allows you to provide early indication to the processor that the receive FIFO is nearly full. A receive FIFO overrun interrupt (ssi_rxo_intr) is generated when the receive shift logic attempts to load data into a completely full receive FIFO. However, this newly received data are lost. A receive FIFO underflow interrupt (ssi_rxu_intr) is generated if

图119。最大
sclk_out/ssi_clk 比率。



仅在正在进行有效传输时，sclk_out线路才会切换。其他时间，该线路保持在非活动状态，状态定义依据其所遵循的串行协议。

sclk_out的频率可由以下公式导出：

$$F_{sclk_out} = \frac{F_{ssi_clk}}{SCKDV}$$

SCKDV是可编程寄存器BAUDR中的位域，取值为0至65534之间的任一偶数。若SCKDV为0，则sclk_out被禁用。

4.10.4.1 频率比摘要

bit率时钟（sclk_out）与DW_apb_ssi外设时钟（ssi_clk）之间频率比的限制摘要如下：

- $F_{ssi_clk} >= 2 \times (\text{maximum } F_{sclk_out})$

4.10.5. 发送与接收 FIFO 缓冲区

DW_apb_ssi使用的FIFO缓冲区为内部16级D型触发器。由于串行规范规定串行传输（数据帧）的长度可为4至16/32位，发送和接收FIFO缓冲区的宽度固定为32位。不足32位的数据帧在写入发送FIFO缓冲区时必须右对齐。移位控制逻辑自动将接收FIFO缓冲区中的接收数据右对齐。

FIFO缓冲区中的每个数据项包含单个数据帧。不可能在单个FIFO位置存储多个数据帧；例如，不能在单个FIFO位置存储两个8位数据帧。若需要8位数据帧，串行移位器传输数据时将忽略FIFO项的高位。

注意

当DW_apb_ssi被禁用（SSI_EN = 0）或复位（presetn）时，发送和接收FIFO缓冲区将被清空。（present）。

通过对DW_apb_ssi数据寄存器（DR）的APB写命令加载发送FIFO。数据由移位控制逻辑从发送FIFO弹出（移除）至发送移位寄存器。当FIFO中条目数小于或等于FIFO阈值时，发送FIFO产生FIFO空中断请求（ssi_txe_intr）。阈值由可编程寄存器TXFTLR设置，决定触发中断的FIFO条目数量。该阈值可提前向处理器指示发送FIFO即将空空。若尝试向已满的发送FIFO写入数据，将触发发送FIFO溢出中断（ssi_txo_intr）。

数据由APB读命令从接收FIFO弹出，读取DW_apb_ssi数据寄存器（DR）。接收FIFO由移位控制逻辑从接收移位寄存器加载。当FIFO中条目数大于或等于阈值加1时，接收FIFO产生FIFO满中断请求（ssi_rxf_intr）。该阈值通过可编程寄存器RXFTLR设置，决定触发中断的FIFO条目数量。

阈值允许您提前向处理器指示接收 FIFO 即将满。当接收移位逻辑尝试将数据加载到已满的接收 FIFO 时，会触发接收 FIFO 溢出中断（ssi_rxo_intr）。然而，此时新接收的数据将丢失。当满足以下条件时，会触发接收 FIFO 欠载中断（ssi_rxu_intr）：

you attempt to read from an empty receive FIFO. This alerts the processor that the read data are invalid.

[Table 577](#) provides description for different Transmit FIFO Threshold values.

Table 577. Transmit FIFO Threshold (TFT) Decode Values

TFT Value	Description
0000_0000	ssi_txe_intr is asserted when zero data entries are present in transmit FIFO
0000_0001	ssi_txe_intr is asserted when one or less data entry is present in transmit FIFO
0000_0010	ssi_txe_intr is asserted when two or less data entries are present in transmit FIFO
...	...
0000_1101	ssi_txe_intr is asserted when 13 or less data entries are present in transmit FIFO
0000_1110	ssi_txe_intr is asserted when 14 or less data entries are present in transmit FIFO
0000_1111	ssi_txe_intr is asserted when 15 or less data entries are present in transmit FIFO

[Table 578](#) provides description for different Receive FIFO Threshold values.

Table 578. Receive FIFO Threshold (RFT) Decode Values

RFT Value	Description
0000_0000	ssi_rxf_intr is asserted when one or more data entry is present in receive FIFO
0000_0001	ssi_rxf_intr is asserted when two or more data entries are present in receive FIFO
0000_0010	ssi_rxf_intr is asserted when three or more data entries are present in receive FIFO
...	...
0000_1101	ssi_rxf_intr is asserted when 14 or more data entries are present in receive FIFO
0000_1110	ssi_rxf_intr is asserted when 15 or more data entries are present in receive FIFO
0000_1111	ssi_rxf_intr is asserted when 16 data entries are present in receive FIFO

4.10.6. 32-Bit Frame Size Support

The IP is configured to set the maximum programmable value in of data frame size to 32 bits. As a result the following features exist:

- dfs_32 (CTRLR0[20:16]) are valid, which contains the value of data frame size. The new register field holds the values 0 to 31. The dfs (CTRLR0[3:0]) is invalid and writing to this register has no effect.
- The receive and transmit FIFO widths are 32 bits.
- All 32 bits of the data register are valid.

4.10.7. SSI Interrupts

The DW_apb_ssi supports combined and individual interrupt requests, each of which can be masked. The combined interrupt request is the ORed result of all other DW_apb_ssi interrupts after masking. Only the combined interrupt request is routed to the Interrupt Controller. All DW_apb_ssi interrupts are level interrupts and are active high.

The DW_apb_ssi interrupts are described as follows:

Transmit FIFO Empty Interrupt (ssi_txe_intr)

Set when the transmit FIFO is equal to or below its threshold value and requires service to prevent an under-run. The threshold value, set through a software-programmable register, determines the level of transmit FIFO entries at which an interrupt is generated. This interrupt is cleared by hardware when data are written into the transmit FIFO buffer, bringing it over the threshold level.

尝试从空的接收 FIFO 读取数据时。此行为提醒处理器读取的数据无效。

[表 577 说明了不同发送 FIFO 阈值的含义。](#)

表 577。发送 FIFO 阈值 (TFT)
解码值

TFT 值	描述
0000_0000	当发送 FIFO 中无数据项时, ssi_txe_intr 被触发
0000_0001	当发送 FIFO 中的数据项少于或等于一项时, ssi_txe_intr 被触发
0000_0010	当发送 FIFO 中的数据项少于或等于两项时, ssi_txe_intr 被触发
...	...
0000_1101	当发送 FIFO 中的数据项少于或等于十三项时, ssi_txe_intr 被触发
0000_1110	当发送 FIFO 中存在不超过 14 个数据条目时, ssi_txe_intr 被置位
0000_1111	当发送 FIFO 中存在不超过 15 个数据条目时, ssi_txe_intr 被置位

[表 578 说明了不同接收 FIFO 阈值的含义。](#)

表 578。接收 FIFO 阈值 (RFT)
解码值

RFT 值	描述
0000_0000	当接收 FIFO 中存在一个或多个数据条目时, ssi_rxf_intr 被置位
0000_0001	当接收 FIFO 中存在两个或多个数据条目时, ssi_rxf_intr 被置位
0000_0010	当接收 FIFO 中存在三个或多个数据条目时, ssi_rxf_intr 被置位
...	...
0000_1101	当接收 FIFO 中存在 14 个或更多数据条目时, ssi_rxf_intr 被置位
0000_1110	当接收 FIFO 中存在 15 个或更多数据条目时, ssi_rxf_intr 被置位
0000_1111	当接收 FIFO 中存在 16 个数据条目时, ssi_rxf_intr 被置位

4.10.6. 32 位帧尺寸支持

该 IP 配置的数据帧大小最大可编程值为 32 位。因此，具备以下特性：

- dfs_32 (CTRLR0[20:16]) 有效，包含数据帧大小的值。该新寄存器字段的取值范围为0至31。dfs (CTRLR0[3:0]) 无效，写入此寄存器无任何效果。
- 接收和发送FIFO的宽度均为32位。
- 数据寄存器的全部32位均为有效。

4.10.7. SSI 中断

DW_apb_ssi支持组合及单独中断请求，且均可屏蔽。组合中断请求为所有其他DW_apb_ssi中断屏蔽后的逻辑或结果。仅组合中断请求被路由至中断控制器。所有DW_apb_ssi中断均为电平中断，且高电平有效。

DW_apb_ssi中断说明如下：

发送FIFO中断 (ssi_txe_intr)

当发送FIFO的填充量等于或低于阈值且需要服务以防止欠载时，该中断被置位。通过软件可编程寄存器设置的阈值确定生成中断的发送 FIFO 条目级别。当数据写入发送 FIFO 缓冲区使其超过阈值时，该中断由硬件清除。

Transmit FIFO Overflow Interrupt (ssi_txo_intr)

Set when an APB access attempts to write into the transmit FIFO after it has been completely filled. When set, data written from the APB is discarded. This interrupt remains set until you read the transmit FIFO overflow interrupt clear register (TXOICR).

Receive FIFO Full Interrupt (ssi_rxf_intr)

Set when the receive FIFO is equal to or above its threshold value plus 1 and requires service to prevent an overflow. The threshold value, set through a software-programmable register, determines the level of receive FIFO entries at which an interrupt is generated. This interrupt is cleared by hardware when data are read from the receive FIFO buffer, bringing it below the threshold level.

Receive FIFO Overflow Interrupt (ssi_rxo_intr)

Set when the receive logic attempts to place data into the receive FIFO after it has been completely filled. When set, newly received data are discarded. This interrupt remains set until you read the receive FIFO overflow interrupt clear register (RXOICR).

Receive FIFO Underflow Interrupt (ssi_rxu_intr)

Set when an APB access attempts to read from the receive FIFO when it is empty. When set, 0s are read back from the receive FIFO. This interrupt remains set until you read the receive FIFO underflow interrupt clear register (RXUICR).

Multi-Master Contention Interrupt (ssi_mst_intr)

Present only when the DW_apb_ssi component is configured as a serial-master device. The interrupt is set when another serial master on the serial bus selects the DW_apb_ssi master as a serial-slave device and is actively transferring data. This informs the processor of possible contention on the serial bus. This interrupt remains set until you read the multi-master interrupt clear register (MSTICR).

Combined Interrupt Request (ssi_intr)

OR'ed result of all the above interrupt requests after masking. To mask this interrupt signal, you must mask all other DW_apb_ssi interrupt requests.

4.10.8. Transfer Modes

When transferring data on the serial bus, the DW_apb_ssi operates in the modes discussed in this section. The transfer mode (TMOD) is set by writing to control register 0 (CTRLR0).

NOTE

The transfer mode setting does not affect the duplex of the serial transfer. TMOD is ignored for Microwire transfers, which are controlled by the MWCR register.

4.10.8.1. Transmit and Receive

When TMOD = **00b**, both transmit and receive logic are valid. The data transfer occurs as normal according to the selected frame format (serial protocol). Transmit data are popped from the transmit FIFO and sent through the txd line to the target device, which replies with data on the rxd line. The receive data from the target device is moved from the receive shift register into the receive FIFO at the end of each data frame.

4.10.8.2. Transmit Only

When TMOD = **01b**, the receive data are invalid and should not be stored in the receive FIFO. The data transfer occurs as normal, according to the selected frame format (serial protocol). Transmit data are popped from the transmit FIFO and sent through the txd line to the target device, which replies with data on the rxd line. At the end of the data frame, the receive shift register does not load its newly received data into the receive FIFO. The data in the receive shift register is

发送 FIFO 溢出中断 (ssi_txo_intr)

当 APB 访问尝试在发送 FIFO 已满后写入数据时触发。设置时，APB 写入的数据将被丢弃。该中断保持设置状态，直到读取发送 FIFO 溢出中断清除寄存器 (TXOICR) 为止。

接收 FIFO 满中断 (ssi_rxf_intr)

当接收 FIFO 达到阈值加 1 或以上时触发，需进行服务以防止溢出。通过软件可编程寄存器设置的阈值确定生成接收 FIFO 中断的条目级别。当从接收 FIFO 缓冲区读取数据使其低于阈值时，该中断由硬件清除。

接收FIFO溢出中断 (ssi_rxo_intr)

当接收逻辑尝试将数据放入已满的接收 FIFO 时，设置该中断。设置时，新接收的数据将被丢弃。该中断将保持设置状态，直到您读取接收 FIFO 溢出中断清除寄存器 (RXOICR)。

接收FIFO欠载中断 (ssi_rxu_intr)

当 APB 访问尝试从空的接收 FIFO 读取数据时，设置该中断。设置时，从接收 FIFO 读出的均为 0。该中断将保持设置状态，直到您读取接收 FIFO 欠载中断清除寄存器 (RXUICR)。

多主控争用中断 (ssi_mst_intr)

仅当 DW_apb_ssi 组件配置为串行主设备时，此中断才存在。当串行总线上的另一串行主设备将 DW_apb_ssi 主控设备选为串行从设备并主动传输数据时，设置该中断。此中断用于通知处理器串行总线可能发生争用。该中断将保持设置状态，直到您读取多主控中断清除寄存器 (MSTICR)。

组合中断请求 (ssi_intr)

屏蔽后所有上述中断请求的或运算结果。要屏蔽此中断信号，必须屏蔽所有其他 DW_apb_ssi 中断请求。

4.10.8. 传输模式

在串行总线上传输数据时，DW_apb_ssi 按本节所述模式运行。传输模式 (TMOD) 通过写入控制寄存器 0 (CTRLR0) 进行设置。

i 注意

传输模式设置不影响串行传输的双工模式。TMOD 对 Microwire 传输无效，Microwire 传输由 MWCR 寄存器控制。

4.10.8.1. 发送与接收

当 TMOD = **00b** 时，发送和接收逻辑均有效。数据传输依据所选帧格式（串行协议）正常进行。发送数据从发送 FIFO 弹出，经由 txd 线发送至目标设备，目标设备通过 rxd 线响应数据。目标设备的接收数据在每个数据帧结束时，从接收移位寄存器移入接收 FIFO。

4.10.8.2. 仅传输

当 TMOD = **01b** 时，接收数据无效，不应存储于接收 FIFO 中。数据传输将根据所选的帧格式（串行协议）正常进行。发送数据从发送 FIFO 弹出，经由 txd 线发送至目标设备，目标设备通过 rxd 线响应数据。在数据帧结束时，接收移位寄存器不会将其新接收的数据加载入接收 FIFO。接收移位寄存器中的数据

overwritten by the next transfer. You should mask interrupts originating from the receive logic when this mode is entered.

4.10.8.3. Receive Only

When TMOD = **10b**, the transmit data are invalid. When configured as a slave, the transmit FIFO is never popped in Receive Only mode. The txd output remains at a constant logic level during the transmission. The data transfer occurs as normal according to the selected frame format (serial protocol). The receive data from the target device is moved from the receive shift register into the receive FIFO at the end of each data frame. You should mask interrupts originating from the transmit logic when this mode is entered.

4.10.8.4. EEPROM Read

NOTE

This transfer mode is only valid for master configurations.

When TMOD = **11b**, the transmit data is used to transmit an opcode and/or an address to the EEPROM device. Typically this takes three data frames (8-bit opcode followed by 8-bit upper address and 8-bit lower address). During the transmission of the opcode and address, no data is captured by the receive logic (as long as the DW_apb_ssi master is transmitting data on its txd line, data on the rxd line is ignored). The DW_apb_ssi master continues to transmit data until the transmit FIFO is empty. Therefore, you should ONLY have enough data frames in the transmit FIFO to supply the opcode and address to the EEPROM. If more data frames are in the transmit FIFO than are needed, then read data is lost.

When the transmit FIFO becomes empty (all control information has been sent), data on the receive line (rxd) is valid and is stored in the receive FIFO; the txd output is held at a constant logic level. The serial transfer continues until the number of data frames received by the DW_apb_ssi master matches the value of the NDF field in the CTRLR1 register + 1.

NOTE

EEPROM read mode is not supported when the DW_apb_ssi is configured to be in the SSP mode.

4.10.9. Operation Modes

The DW_apb_ssi can be configured in the fundamental modes of operation discussed in this section.

4.10.9.1. Serial Master Mode

This mode enables serial communication with serial-slave peripheral devices. When configured as a serial-master device, the DW_apb_ssi initiates and controls all serial transfers. [Figure 120](#) shows an example of the DW_apb_ssi configured as a serial master with all other devices on the serial bus configured as serial slaves.

将被下一次传输覆盖。进入该模式时，应屏蔽来自接收逻辑的中断。

4.10.8.3. 仅接收

当 TMOD = **10b** 时，发送数据无效。作为从机配置时，发送 FIFO 在仅接收模式下不会弹出数据。传输过程中，txd 输出保持恒定逻辑电平。数据传输依据所选帧格式（串行协议）正常进行。目标设备的接收数据在每个数据帧结束时由接收移位寄存器移入接收 FIFO。进入该模式时，应屏蔽来自发送逻辑的中断。

4.10.8.4. EEPROM 读取

i 注意

此传输模式仅适用于主控配置。

当 TMOD = **11b** 时，传输的数据用于向 EEPROM 设备发送操作码和/或地址。通常这需要三个数据帧（8 位操作码，随后是 8 位高地址字节和 8 位低地址字节）。在发送操作码和地址期间，接收逻辑不会捕获任何数据（只要 DW_apb_ssi 主控通过其 txd 线发送数据，rxd 线上的数据即被忽略）。DW_apb_ssi 主控会持续发送数据，直到发送 FIFO 为空。因此，发送 FIFO 中应仅存放足够发送操作码和地址的数据帧；如果发送 FIFO 中的数据帧超过所需数量，读取数据将会丢失。

当发送 FIFO 变为空（所有控制信息已发送完毕）后，接收线（rxd）上的数据变为有效并存储至接收 FIFO；txd 输出保持在恒定的逻辑电平。串行传输将持续进行，直到 DW_apb_ssi 主控接收到的数据帧数量与 CTRLR1 寄存器中 NDF 字段的值加 1 相匹配。

i 注意

当 DW_apb_ssi 配置为 SSP 模式时，不支持 EEPROM 读取模式。

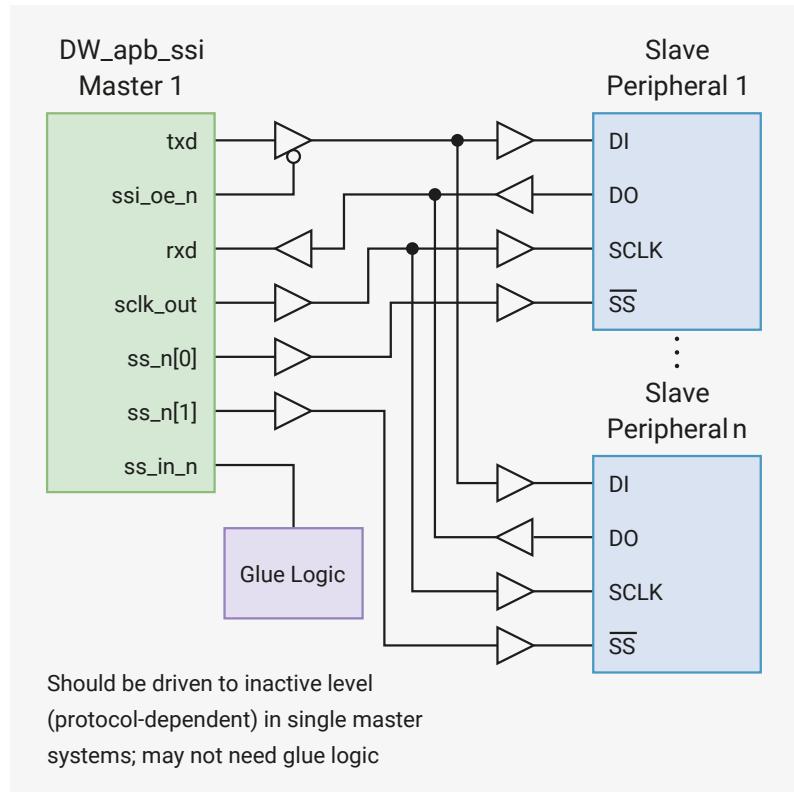
4.10.9. 操作模式

DW_apb_ssi 可配置为本节所述的基本操作模式。

4.10.9.1 串行主控模式

该模式支持与串行从属外设进行串行通信。当配置为串行主控设备时，DW_apb_ssi 负责发起并控制所有串行传输。图 12-0 展示了 DW_apb_ssi 配置为串行主控，且串行总线上的所有其他设备均配置为串行从属的示例。

Figure 120.
DW_apb_ssi
Configured as Master
Device



The serial bit-rate clock, generated and controlled by the DW_apb_ssi, is driven out on the sclk_out line. When the DW_apb_ssi is disabled ($\text{SSI_EN} = 0$), no serial transfers can occur and sclk_out is held in “inactive” state, as defined by the serial protocol under which it operates.

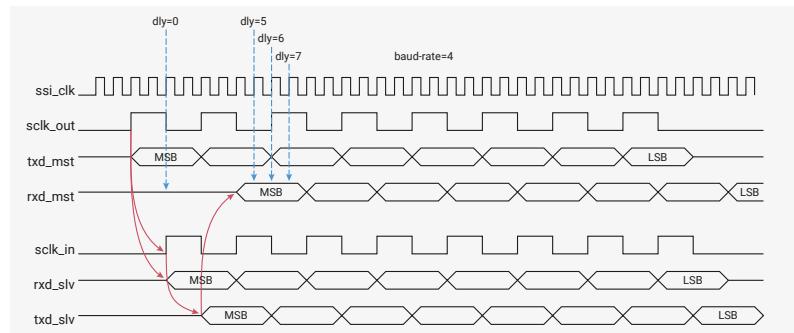
Multiple master configuration is not supported.

4.10.9.1.1. RXD Sample Delay

When the DW_apb_ssi is configured as a master, additional logic can be included in the design in order to delay the default sample time of the rxd signal. This additional logic can help to increase the maximum achievable frequency on the serial bus.

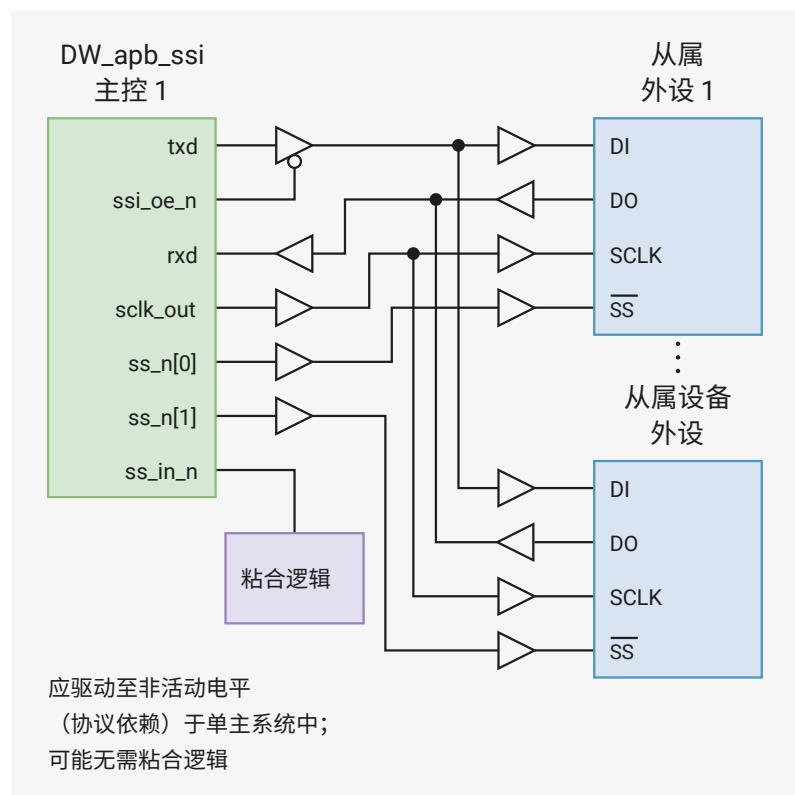
Round trip routing delays on the sclk_out signal from the master and the rxd signal from the slave can mean that the timing of the rxd signal—as seen by the master—has moved away from the normal sampling time. [Figure 121](#) illustrates this situation.

Figure 121. Effects of
Round-Trip Routing
Delays on sclk_out
Signal



The Slave uses the sclk_out signal from the master as a strobe in order to drive rxd signal data onto the serial bus. Routing and sampling delays on the sclk_out signal by the slave device can mean that the rxd bit has not stabilized to the correct value before the master samples the rxd signal. [Figure 121](#) shows an example of how a routing delay on the rxd signal can result in an incorrect rxd value at the default time when the master samples the port.

图120.
DW_apb_ssi
*i*配置为主设备



串行比特率时钟由DW_apb_ssi生成和控制，并通过sclk_out线输出。当DW_apb_ssi禁用（SSI_EN = 0）时，不允许进行串行传输，且sclk_out保持在所用串行协议定义的“非活动”状态。

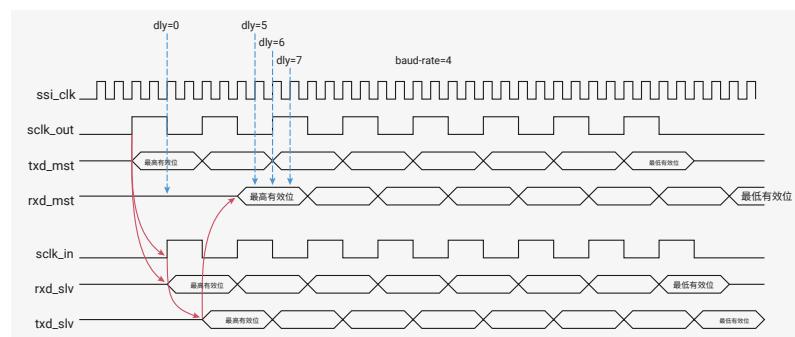
不支持多主配置。

4.10.9.1.1 RXD采样延迟

当DW_apb_ssi配置为主设备时，设计中可加入附加逻辑以延迟rxn信号的默认采样时间。该附加逻辑有助于提高串行总线的最大可实现频率。

主控设备的sclk_out信号与从属设备的rxn信号之间存在的往返路由延迟，可能导致主控设备看到的rxn信号时序偏离正常采样时间。图121示意了该情况。

图121。往返路由
延迟对sclk_out
信号的影响



从属设备使用主控设备的sclk_out信号作为触发信号，以驱动rxn信号数据传输至串行总线。

从属设备对sclk_out信号的路由及采样延迟，可能导致rxn位在主控设备采样该信号之前尚未稳定至正确值。图121展示了rxn信号路由延迟如何导致主控设备在默认采样时间采样到错误rxn值的示例。

Without the RXD Sample Delay logic, the user would have to increase the baud-rate for the transfer in order to ensure that the setup times on the rxd signal are within range; this results in reducing the frequency of the serial interface.

When the RXD Sample Delay logic is included, the user can dynamically program a delay value in order to move the sampling time of the rxd signal equal to a number of ssi_clk cycles from the default.

The sample delay logic has a resolution of one ssi_clk cycle. Software can “train” the serial bus by coding a loop that continually reads from the slave and increments the master’s RXD Sample Delay value until the correct data is received by the master.

4.10.9.1.2. Data Transfers

Data transfers are started by the serial-master device. When the DW_apb_ssi is enabled (`SSI_EN=1`), at least one valid data entry is present in the transmit FIFO and a serial-slave device is selected. When actively transferring data, the busy flag (`BUSY`) in the status register (`SR`) is set. You must wait until the busy flag is cleared before attempting a new serial transfer.

 **NOTE**

The `BUSY` status is not set when the data are written into the transmit FIFO. This bit gets set only when the target slave has been selected and the transfer is underway. After writing data into the transmit FIFO, the shift logic does not begin the serial transfer until a positive edge of the `sclk_out` signal is present. The delay in waiting for this positive edge depends on the baud rate of the serial transfer. Before polling the `BUSY` status, you should first poll the TFE status (waiting for 1) or wait for $\text{BAUDR} * \text{ssi_clk}$ clock cycles.

4.10.9.1.3. Master SPI and SSP Serial Transfers

When the transfer mode is “transmit and receive” or “transmit only” (`TMOD = 00b` or `TMOD = 01b`, respectively), transfers are terminated by the shift control logic when the transmit FIFO is empty. For continuous data transfers, you must ensure that the transmit FIFO buffer does not become empty before all the data have been transmitted. The transmit FIFO threshold level (`TXFTLR`) can be used to early interrupt (`ssi_txe_intr`) the processor indicating that the transmit FIFO buffer is nearly empty. When a DMA is used for APB accesses, the transmit data level (`DMATDLR`) can be used to early request (`dma_tx_req`) the DMA Controller, indicating that the transmit FIFO is nearly empty. The FIFO can then be refilled with data to continue the serial transfer. The user may also write a block of data (at least two FIFO entries) into the transmit FIFO before enabling a serial slave. This ensures that serial transmission does not begin until the number of data-frames that make up the continuous transfer are present in the transmit FIFO.

When the transfer mode is “receive only” (`TMOD = 10b`), a serial transfer is started by writing one “dummy” data word into the transmit FIFO when a serial slave is selected. The `txd` output from the DW_apb_ssi is held at a constant logic level for the duration of the serial transfer. The transmit FIFO is popped only once at the beginning and may remain empty for the duration of the serial transfer. The end of the serial transfer is controlled by the “number of data frames” (`NDF`) field in control register 1 (`CTRLR1`).

If, for example, you want to receive 24 data frames from a serial-slave peripheral, you should program the `NDF` field with the value 23; the receive logic terminates the serial transfer when the number of frames received is equal to the `NDF` value + 1. This transfer mode increases the bandwidth of the APB bus as the transmit FIFO never needs to be serviced during the transfer. The receive FIFO buffer should be read each time the receive FIFO generates a FIFO full interrupt request to prevent an overflow.

When the transfer mode is “eeprom_read” (`TMOD = 11b`), a serial transfer is started by writing the opcode and/or address into the transmit FIFO when a serial slave (EEPROM) is selected. The opcode and address are transmitted to the EEPROM device, after which read data is received from the EEPROM device and stored in the receive FIFO. The end of the serial transfer is controlled by the `NDF` field in the control register 1 (`CTRLR1`).

若无 RXD 采样延迟逻辑，用户需提高传输波特率，以确保 rxd 信号的建立时间处于可接受范围内；这会导致串行接口频率降低。

当包含 RXD 采样延迟逻辑时，用户可以动态设置延迟值，以将 rxd 信号的采样时间从默认位置移动若干 ssi_clk 时钟周期。

采样延迟逻辑的分辨率为一个 ssi_clk 时钟周期。软件可通过编写循环程序，不断从从设备读取数据并递增主设备的 RXD 采样延迟值，从而“训练”串行总线，直到主设备正确接收数据。

4.10.9.1.2. 数据传输

数据传输由串行主设备发起。当 DW_apb_ssi 启用 (`SSI_EN=1`)，传输 FIFO 中至少存在一条有效数据，且已选择串行从设备时。在主动传输数据时，状态寄存器 (SR) 中的忙标志 (BUSY) 被置位。必须等待忙标志清除后，方可尝试新一轮串行传输。

注意

当数据写入发送FIFO时，BUSY状态不会被置位。仅当目标从设备被选中且传输正在进行时，该位才会被置位。在向发送FIFO写入数据后，移位逻辑不会开始串行传输，直到检测到sclk_out信号的上升沿。等待该上升沿的延迟取决于串行传输的波特率。在查询BUSY状态之前，应先查询TFE状态（等待其为1）或等待BAUDR * ssi_clk个时钟周期。

4.10.9.1.3. 主SPI和SSP串行传输

当传输模式为“发送并接收”或“仅发送”（分别对应`TMOD = 00b`或`TMOD = 01b`）时，移位控制逻辑将在发送FIFO为空时终止传输。对于连续数据传输，必须确保在所有数据传输完成之前，发送FIFO缓冲区不会变为空。传输FIFO阈值级别 (TXFT LR) 可用于提前中断处理器 (`ssi_txe_intr`)，以指示传输FIFO缓冲区接近空闲状态。当APB访问采用DMA时，传输数据级别 (DMATDLR) 可用于提前请求DMA控制器 (`dma_tx_req`)，以表明传输FIFO接近空闲状态。随后可以用数据重新填充FIFO，以继续串行传输。用户也可在启用串行从设备之前，向传输FIFO写入至少两个FIFO条目的数据块。这确保在传输FIFO中积累足够构成连续传输的数据帧数量之前，串行传输不会启动。

当传输模式设为“仅接收”（`TMOD = 10b`）时，选择串行从设备时通过向传输FIFO写入一个“虚拟”数据字以启动串行传输。`DW_apb_ssi`的txd输出在整个串行传输过程中保持恒定逻辑电平。发送FIFO仅在开始时弹出一次，且可能在整个串行传输过程中保持为空。串行传输的结束由控制寄存器1 (CTRLR1) 中的“数据帧数” (NDF) 字段决定。

例如，若您希望从串行从设备接收24个数据帧，应将NDF字段设为23；当接收的数据帧数等于NDF值加1时，接收逻辑即终止串行传输。此传输模式提升了APB总线的带宽，因为传输期间无需为发送FIFO提供服务。每当接收FIFO产生FIFO满中断请求时，必须及时读取接收FIFO缓冲区以防止溢出。

当传输模式为“eeprom_read”（`TMOD = 11b`）时，选定串行从设备（EEPROM）后，通过写入操作码和/或地址至发送FIFO启动串行传输。操作码和地址传输至EEPROM设备，随后从EEPROM设备接收数据并存储于接收FIFO中。串行传输的结束由控制寄存器1 (CTRLR1) 中的NDF字段控制。

NOTE

EEPROM read mode is not supported when the DW_apb_ssi is configured to be in the SSP mode.

The receive FIFO threshold level (RXFTLR) can be used to give early indication that the receive FIFO is nearly full. When a DMA is used for APB accesses, the receive data level (DMARDLR) can be used to early request (dma_rx_req) the DMA Controller, indicating that the receive FIFO is nearly full.

A typical software flow for completing an SPI or SSP serial transfer from the DW_apb_ssi serial master is outlined as follows:

1. If the DW_apb_ssi is enabled, disable it by writing 0 to the SSI Enable register (SSIENR).
2. Set up the DW_apb_ssi control registers for the transfer; these registers can be set in any order.
 - o Write Control Register 0 (CTRLR0). For SPI transfers, the serial clock polarity and serial clock phase parameters must be set identical to target slave device.
 - o If the transfer mode is receive only, write CTRLR1 (Control Register 1) with the number of frames in the transfer minus 1; for example, if you want to receive four data frames, if you want to receive four data frames, write '3' into CTRLR1.
 - o Write the Baud Rate Select Register (BAUDR) to set the baud rate for the transfer.
 - o Write the Transmit and Receive FIFO Threshold Level registers (TXFTLR and RXFTLR, respectively) to set FIFO threshold levels.
 - o Write the IMR register to set up interrupt masks.
 - o The Slave Enable Register (SER) register can be written here to enable the target slave for selection. If a slave is enabled here, the transfer begins as soon as one valid data entry is present in the transmit FIFO. If no slaves are enabled prior to writing to the Data Register (DR), the transfer does not begin until a slave is enabled.
3. Enable the DW_apb_ssi by writing 1 to the SSIENR register.
4. Write data for transmission to the target slave into the transmit FIFO (write DR). If no slaves were enabled in the SER register at this point, enable it now to begin the transfer.
5. Poll the BUSY status to wait for completion of the transfer. The BUSY status cannot be polled immediately.
6. If a transmit FIFO empty interrupt request is made, write the transmit FIFO (write DR). If a receive FIFO full interrupt request is made, read the receive FIFO (read DR).
7. The transfer is stopped by the shift control logic when the transmit FIFO is empty. If the transfer mode is receive only (TMOD = **10b**), the transfer is stopped by the shift control logic when the specified number of frames have been received. When the transfer is done, the BUSY status is reset to 0.
8. If the transfer mode is not transmit only (TMOD != **01b**), read the receive FIFO until it is empty.
9. Disable the DW_apb_ssi by writing 0 to SSIENR.

[Figure 122](#) shows a typical software flow for starting a DW_apb_ssi master SPI/SSP serial transfer. The diagram also shows the hardware flow inside the serial-master component.

注意

当 DW_apb_ssi 配置为 SSP 模式时，不支持 EEPROM 读取模式。

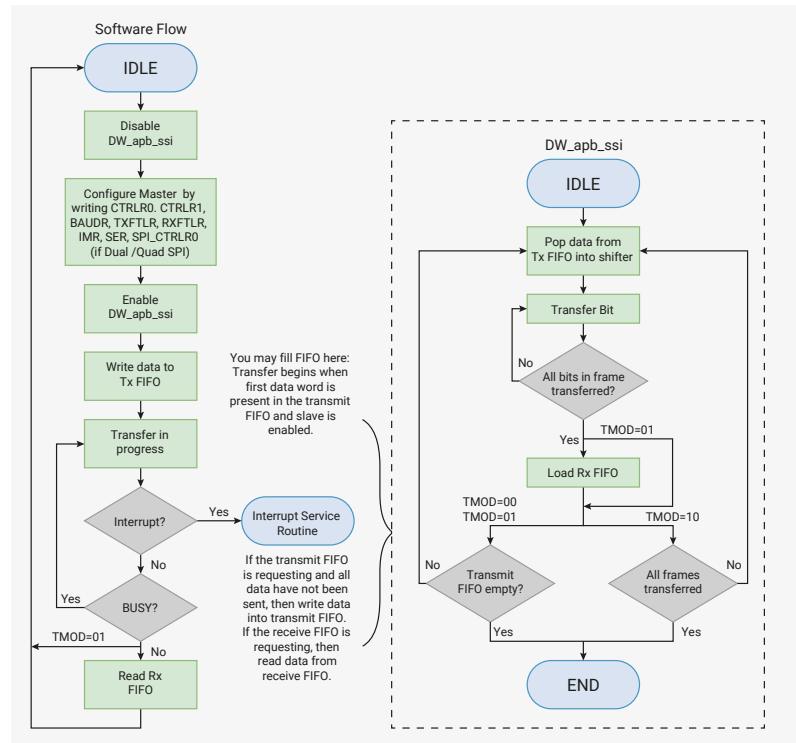
接收FIFO阈值水平（RXFTLR）可用于提前指示接收FIFO即将满载。当APB访问使用DMA时，接收数据水平（DMARDLR）可用于提前请求（dma_rx_req）DMA控制器，指示接收FIFO即将满载。

完成来自DW_apb_ssi串行主设备的SPI或SSP串行传输的典型软件流程如下所示：

1. 若DW_apb_ssi已启用，应通过向SSI使能寄存器（SSIENR）写入0以禁用它。
2. 配置DW_apb_ssi的传输控制寄存器；这些寄存器可按任意顺序设置。
 - 写入控制寄存器0（CTRLR0）。对于SPI传输，必须将串行时钟极性和串行时钟相位参数设置为与目标从机设备一致。
 - 如果传输模式为仅接收，则将CTRLR1（控制寄存器1）写为传输帧数减一；例如，若要接收四个数据帧，则在CTRLR1中写入“3”。
 - 写入波特率选择寄存器（BAUDR）以设置传输波特率。
 - 写入发送和接收FIFO阈值寄存器（分别为TXFTLR和RXFTLR）以设置FIFO阈值。
 - 写入IMR寄存器以设置中断屏蔽。
 - 此处可写入从机使能寄存器（SER）以启用目标从机设备的选通。若此处启用从机，则只要发送FIFO中存在一个有效数据项，传输即开始。若在写入数据寄存器（DR）之前未启用任何从机，传输将在启用从机后开始。
3. 通过向 SSIENR 寄存器写入 1 以启用 DW_apb_ssi。
4. 将要传输的数据写入发送 FIFO（写 DR）以发送至目标从设备。若此时 SER 寄存器中未启用任何从设备，请立即启用以开始传输。
5. 轮询 BUSY 状态以等待传输完成。BUSY 状态不可立即轮询。
6. 若发生发送 FIFO 空中断请求，写入发送 FIFO（写 DR）；若发生接收 FIFO 满中断请求，读取接收 FIFO（读 DR）。
7. 当发送 FIFO 为空时，移位控制逻辑会停止传输。若传输模式为仅接收（TMOD = **10b**），当接收指定数量的帧后，移位控制逻辑会停止传输。传输完成后，BUSY 状态被重置为 0。
8. 若传输模式非仅发送（TMOD != **01b**），则读取接收 FIFO 直至为空。
9. 通过向SSIENR寄存器写入0以禁用DW_apb_ssi。

图122展示了启动DW_apb_ssi主控SPI/SSP串行传输的典型软件流程。该图亦显示了串行主控组件内部的硬件流程。

Figure 122.

DW_apb_ssi Master
SPI/SSI Transfer Flow

4.10.9.1.4. Master Microwire Serial Transfers

Microwire serial transfers from the DW_apb_ssi serial master are controlled by the Microwire Control Register (MWCR). The MWHS bit field enables and disables the Microwire handshaking interface. The MDD bit field controls the direction of the data frame (the control frame is always transmitted by the master and received by the slave). The MWMOD bit field defines whether the transfer is sequential or nonsequential.

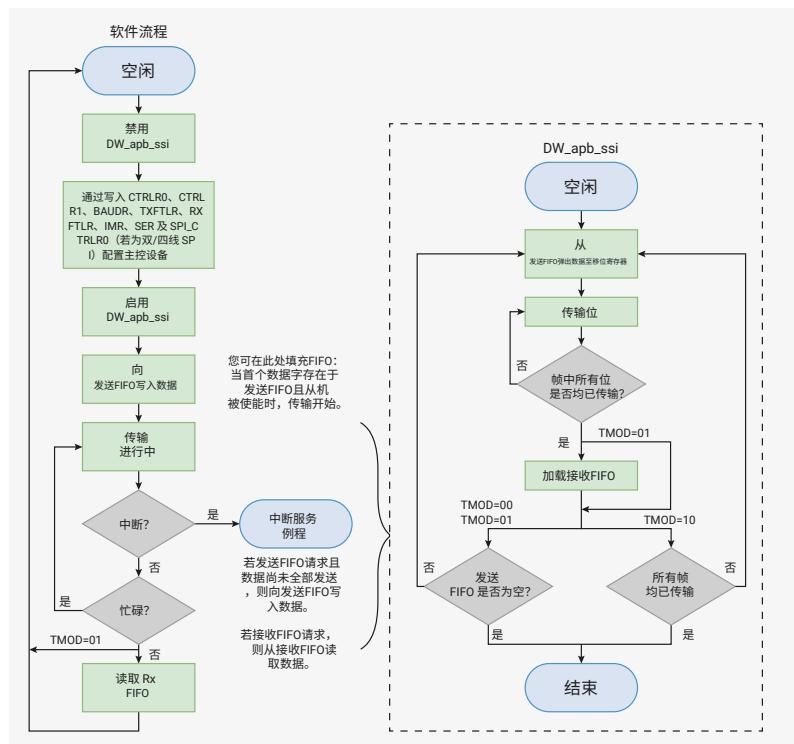
All Microwire transfers are started by the DW_apb_ssi serial master when there is at least one control word in the transmit FIFO and a slave is enabled. When the DW_apb_ssi master transmits the data frame (MDD = 1), the transfer is terminated by the shift logic when the transmit FIFO is empty. When the DW_apb_ssi master receives the data frame (MDD = 1), the termination of the transfer depends on the setting of the MWMOD bit field. If the transfer is nonsequential (MWMOD = 0), it is terminated when the transmit FIFO is empty after shifting in the data frame from the slave. When the transfer is sequential (MWMOD = 1), it is terminated by the shift logic when the number of data frames received is equal to the value in the CTRLR1 register + 1.

When the handshaking interface on the DW_apb_ssi master is enabled (MWHS = 1), the status of the target slave is polled after transmission. Only when the slave reports a ready status does the DW_apb_ssi master complete the transfer and clear its BUSY status. If the transfer is continuous, the next control/data frame is not sent until the slave device returns a ready status.

A typical software flow for completing a Microwire serial transfer from the DW_apb_ssi serial master is outlined as follows:

1. If the DW_apb_ssi is enabled, disable it by writing 0 to SSIENR.
2. Set up the DW_apb_ssi control registers for the transfer. These registers can be set in any order. Write CTRLR0 to set transfer parameters.
 - o If the transfer is sequential and the DW_apb_ssi master receives data, write CTRLR1 with the number of frames in the transfer minus 1; for instance, if you want to receive four data frames, write '3' into CTRLR1.
 - o Write BAUDR to set the baud rate for the transfer.
 - o Write TXFTLR and RXFTLR to set FIFO threshold levels.
 - o Write the IMR register to set up interrupt masks.

图 122。
DW_apb_ssi 主控设备
SPI/SSP 传输流程



4.10.9.1.4. 主控 Microwire 串行传输

DW_apb_ssi 串行主控设备的 Microwire 串行传输由 Microwire 控制寄存器 (MWCR) 控制。

MWHS 位域用于启用或禁用 Microwire 握手接口。MDD 位域控制数据帧方向（控制帧始终由主控设备发送，由从属设备接收）。MWMOD 位域定义传输模式为顺序或非顺序。

当传输 FIFO 中至少有一个控制字且从设备被使能时，DW_apb_ssi 串行主设备将启动所有 Microwire 传输。当 DW_apb_ssi 主设备发送数据帧 (MDD = 1) 时，移位逻辑将在传输 FIFO 为空时终止传输。当 DW_apb_ssi 主设备接收数据帧 (MD D = 1) 时，传输的终止取决于 MWMOD 位字段的设置。如果传输为非连续传输 (MWMOD = 0)，则在从从机移入数据帧后且传输 FIFO 为空时终止传输。当传输为连续传输 (MWMOD = 1) 时，移位逻辑将在接收的数据帧数等于 CTRLRO 寄存器中的值加 1 时终止传输。

当 DW_apb_ssi 主设备上的握手接口被使能 (MWHS = 1) 时，传输完成后将轮询目标从机的状态。仅当从设备报告处于就绪状态时，DW_apb_ssi 主设备才完成传输并清除其 BUSY 状态。若传输为连续模式，则在从设备返回就绪状态之前，不发送下一控制或数据帧。

完成 DW_apb_ssi 串行主设备的 Microwire 串行传输的软件典型流程如下：

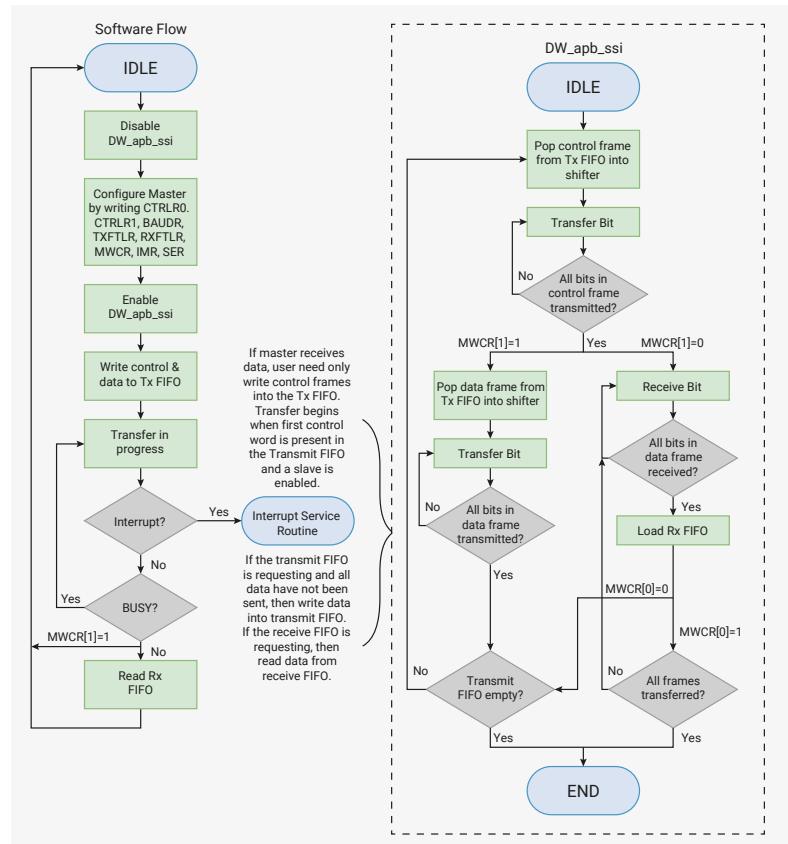
1. 若 DW_apb_ssi 已启用，则通过向 SSIENR 寄存器写入 0 来禁用该模块。
2. 配置 DW_apb_ssi 控制寄存器以准备传输。上述寄存器可以按任意顺序配置。向 CTRLRO 寄存器写入数据以设置传输参数。
 - 当传输为顺序模式且 DW_apb_ssi 主设备接收数据时，向 CTRLR1 寄存器写入传输帧数减一的值；例如，若要接收四帧数据，应向 CTRLR1 写入“3”。
 - 向 BAUDR 寄存器写入数据以设置传输波特率。
 - 向 TXFTLR 和 RXFTLR 寄存器写入数据以设置 FIFO 阈值电平。
 - 写入 IMR 寄存器以设置中断屏蔽。

You can write the SER register to enable the target slave for selection. If a slave is enabled here, the transfer begins as soon as one valid data entry is present in the transmit FIFO. If no slaves are enabled prior to writing to the DR register, the transfer does not begin until a slave is enabled.

3. Enable the DW_apb_ssi by writing 1 to the SSIENR register.
 4. If the DW_apb_ssi master transmits data, write the control and data words into the transmit FIFO (write DR). If the DW_apb_ssi master receives data, write the control word(s) into the transmit FIFO.
- If no slaves were enabled in the SER register at this point, enable now to begin the transfer.
5. Poll the BUSY status to wait for completion of the transfer. The BUSY status cannot be polled immediately.
 6. The transfer is stopped by the shift control logic when the transmit FIFO is empty. If the transfer mode is sequential and the DW_apb_ssi master receives data, the transfer is stopped by the shift control logic when the specified number of data frames is received. When the transfer is done, the BUSY status is reset to 0.
 7. If the DW_apb_ssi master receives data, read the receive FIFO until it is empty.
 8. Disable the DW_apb_ssi by writing 0 to SSIENR.

Figure 123 shows a typical software flow for starting a DW_apb_ssi master Microwire serial transfer. The diagram also shows the hardware flow inside the serial-master component.

Figure 123.
DW_apb_ssi Master
Microwire Transfer
Flow



4.10.10. Partner Connection Interfaces

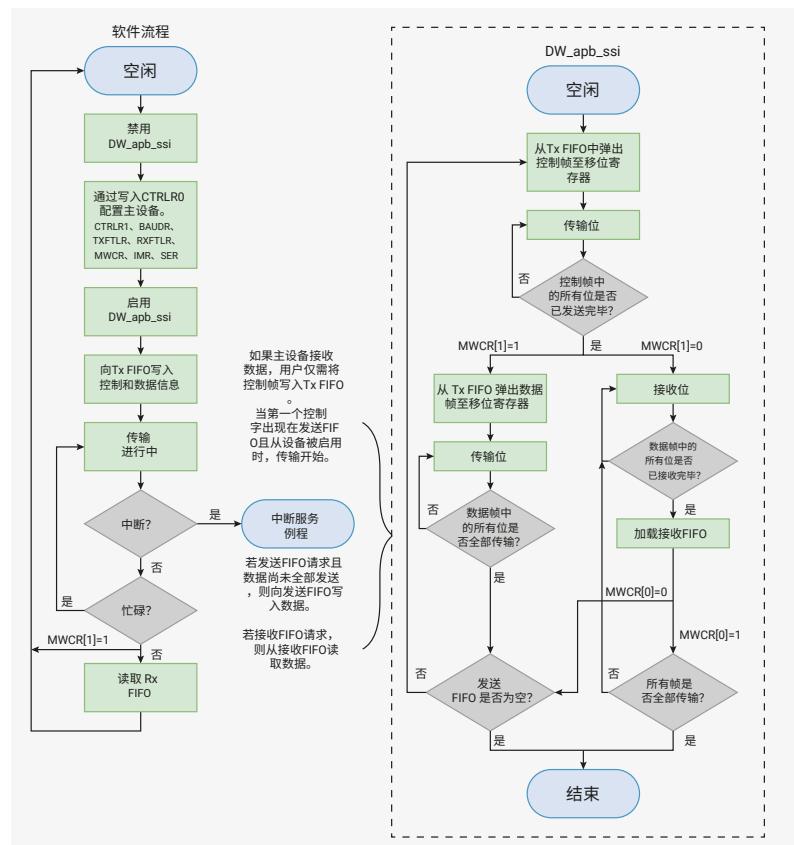
The DW_apb_ssi can connect to any serial-slave peripheral device using one of the interfaces discussed in the following sections.

您可以写入SER寄存器以使目标从设备被选中。如果此处启用了某个从设备，只要传输FIFO中有一个有效数据项，传输即开始。如果在写入DR寄存器之前未启用任何从设备，则传输将在启用从设备后才开始。

3. 通过向SSIENR寄存器写入1以启用DW_apb_ssi。
 4. 若DW_apb_ssi主设备发送数据，应将控制字和数据字写入传输FIFO（写入DR）。若DW_apb_ssi主设备接收数据，应将控制字写入传输FIFO。
- 若此时SER寄存器中未启用任何从设备，请立即启用以开始传输。
5. 轮询BUSY状态以等待传输完成。BUSY状态不可立即轮询。
 6. 当发送FIFO为空时，移位控制逻辑会停止传输。当传输模式为顺序且DW_apb_ssi主设备接收数据时，移位控制逻辑将在接收指定数量的数据帧后停止传输。传输完成后，BUSY状态被重置为0。
7. 若DW_apb_ssi主设备接收数据，需读取接收FIFO直至其为空。
 8. 通过向SSIENR寄存器写入0以禁用DW_apb_ssi。

图123展示了启动DW_apb_ssi主设备Microwire串行传输的典型软件流程。该图亦显示了串行主控组件内部的硬件流程。

图123。
DW_apb_ssi主控设备
Microwire传输
流



4.10.10. 对端连接接口

DW_apb_ssi可通过以下章节中讨论的任一接口连接至任何串行从属外设。

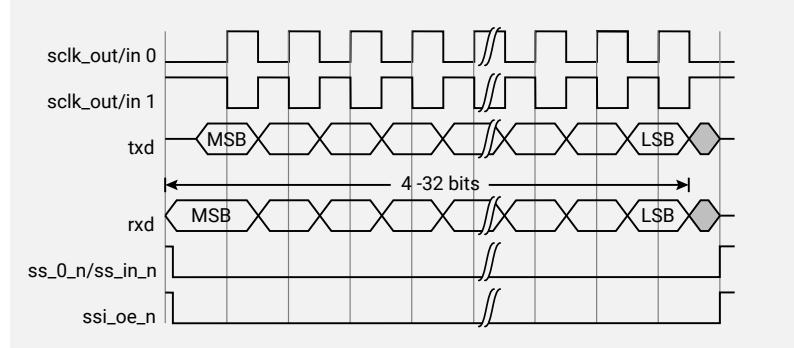
4.10.10.1. Motorola Serial Peripheral Interface (SPI)

With the SPI, the clock polarity (SCPOL) configuration parameter determines whether the inactive state of the serial clock is high or low. To transmit data, both SPI peripherals must have identical serial clock phase (SCPH) and clock polarity (SCPOL) values. The data frame can be 4 to 16/32 bits (depending upon SSI_MAX_XFER_SIZE) in length.

When the configuration parameter SCPH = 0, data transmission begins on the falling edge of the slave select signal. The first data bit is captured by the master and slave peripherals on the first edge of the serial clock; therefore, valid data must be present on the txd and rxd lines prior to the first serial clock edge.

Figure 124 shows a timing diagram for a single SPI data transfer with SCPH = 0. The serial clock is shown for configuration parameters SCPOL = 0 and SCPOL = 1.

Figure 124. SPI Serial Format (SCPH = 0)



The following signals are illustrated in the timing diagrams in this section:

sclk_out

serial clock from DW_apb_ssi master

ss_0_n

slave select signal from DW_apb_ssi master

ss_in_n

slave select input to the DW_apb_ssi slave

ss_oe_n

output enable for the DW_apb_ssi master

txd

transmit data line for the DW_apb_ssi master

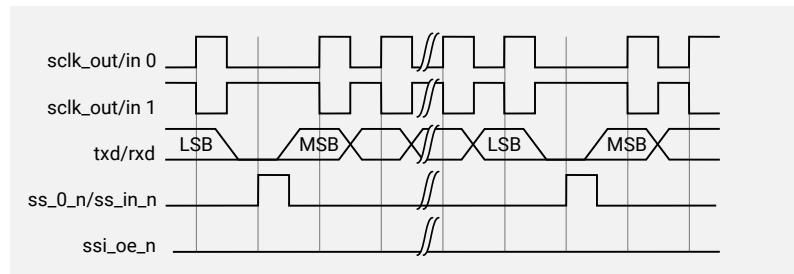
rxd

receive data line for the DW_apb_ssi master

Continuous data transfers are supported when SCPH = 0:

- When CTRLR0.SSTE is set to 1, the DW_apb_ssi toggles the slave select signal between frames and the serial clock is held to its default value while the slave select signal is active; this operating mode is illustrated in [Figure 125](#).

Figure 125. Serial Format Continuous Transfers (SCPH = 0)



When the configuration parameter SCPH = 1, master peripherals begin transmitting data on the first serial clock edge

4.10.10.1. 摩托罗拉串行外设接口（SPI）

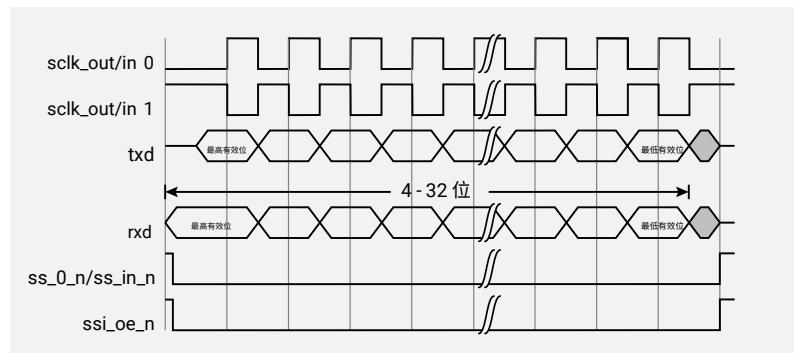
使用 SPI 时，时钟极性（SCPOL）配置参数决定串行时钟的非活动状态为高电平还是低电平。要传输数据，两个 SPI 外设必须具有相同的串行时钟相位（SCPH）和时钟极性（SCPOL）值。数据帧长度可为 4 至 16/32 位（具体取决于 SSI_MA_X_XFER_SIZE）。

当配置参数 SCPH=0 时，数据传输自从属选择信号的下降沿开始。

第一个数据位由主设备和从设备在串行时钟的第一个沿捕获；因此，有效数据须在第一个串行时钟沿到来之前存在于 txd 和 rxd 线路上。

图 124 显示了 SCPH=0 状态下单次 SPI 数据传输的时序图。序列时钟示意了配置参数 SCPOL = 0 及 SCPOL = 1 的情况。

图 124。SPI 串行格式 (SCPH = 0)



本节时序图中所示信号包括：

sclk_out

来自 DW_apb_ssi 主控的串行时钟

ss_0_n

来自 DW_apb_ssi 主控的从机选择信号

ss_in_n

输入至 DW_apb_ssi 从机的从机选择信号

ssi_oe_n

DW_apb_ssi 主控的输出使能信号

txd

DW_apb_ssi 主控的发送数据线

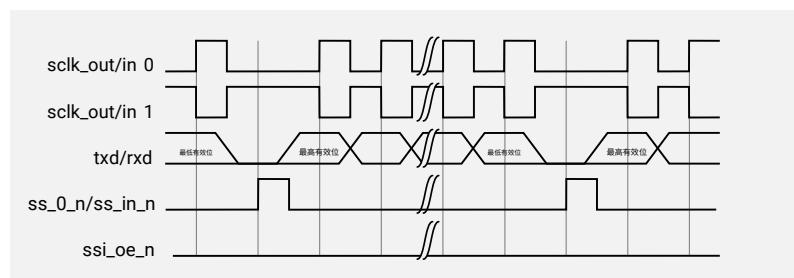
rxn

DW_apb_ssi 主控的接收数据线

当 SCPH = 0 时，支持连续数据传输：

- 当 CTRL0.SSTE 设为 1 时，DW_apb_ssi 会在帧间切换从机选择信号，且在从机选择信号有效期间，串行时钟保持默认值；该运行模式如图125所示。

图125。串行
格式连续
传输 (SCPH = 0)

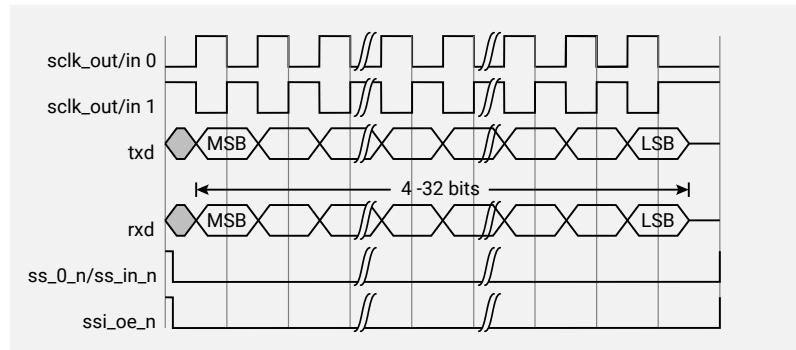


当配置参数 SCPH = 1 时，主设备外设在从选择线激活后的第一个串行时钟边沿开始传输数据。

after the slave select line is activated. The first data bit is captured on the second (trailing) serial clock edge. Data are propagated by the master peripherals on the leading edge of the serial clock. During continuous data frame transfers, the slave select line may be held active-low until the last bit of the last frame has been captured.

[Figure 126](#) shows the timing diagram for the SPI format when the configuration parameter SCPH = 1.

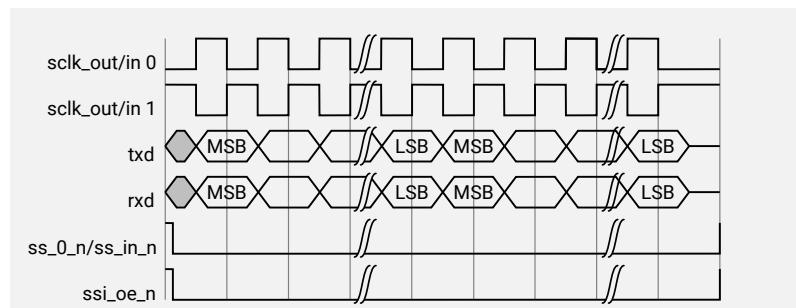
Figure 126. SPI Serial Format (SCPH = 1)



Continuous data frames are transferred in the same way as single frames, with the MSB of the next frame following directly after the LSB of the current frame. The slave select signal is held active for the duration of the transfer.

[Figure 127](#) shows the timing diagram for continuous SPI transfers when the configuration parameter SCPH = 1.

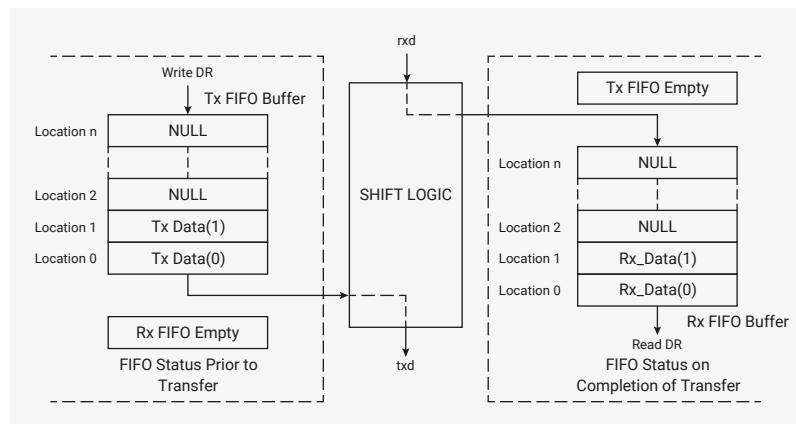
Figure 127. SPI Serial Format Continuous Transfer (SCPH = 1)



There are four possible transfer modes on the DW_apb_ssi for performing SPI serial transactions. For transmit and receive transfers (transfer mode field (9:8) of the Control Register 0 = **00b**), data transmitted from the DW_apb_ssi to the external serial device is written into the transmit FIFO. Data received from the external serial device into the DW_apb_ssi is pushed into the receive FIFO.

[Figure 128](#) shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are transmitted from the DW_apb_ssi to the external serial device in a continuous transfer. The external serial device also responds with two data words for the DW_apb_ssi.

Figure 128. FIFO Status for Transmit & Receive SPI and SSP Transfers

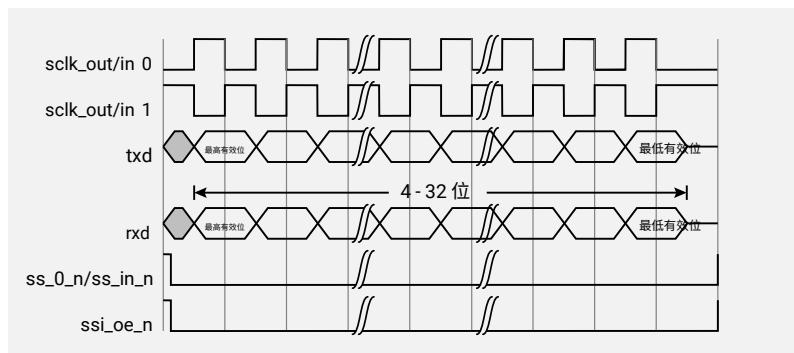


For transmit only transfers (transfer mode field (9:8) of the Control Register 0 = **01b**), data transmitted from the DW_apb_ssi to the external serial device is written into the transmit FIFO. As the data received from the external serial device is deemed invalid, it is not stored in the DW_apb_ssi receive FIFO.

从选择线激活后。第一个数据位在第二个（后沿）串行时钟边沿被捕获。数据由主设备外设在串行时钟的前沿传播。在连续数据帧传输期间，从选择线可保持低电平，直到最后一帧的最后一位被捕获。

图126展示了配置参数SCPH = 1时的SPI格式时序图。

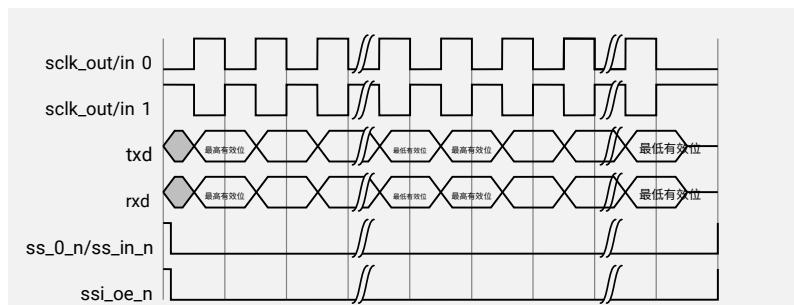
图126。 SPI串行
格式 (SCPH = 1)



连续数据帧的传输方式与单帧相同，下一帧的最高有效位紧接当前帧的最低有效位之后传输。从设备选择信号在整个传输过程中保持有效。

图127展示了配置参数SCPH = 1时连续SPI传输的时序图。

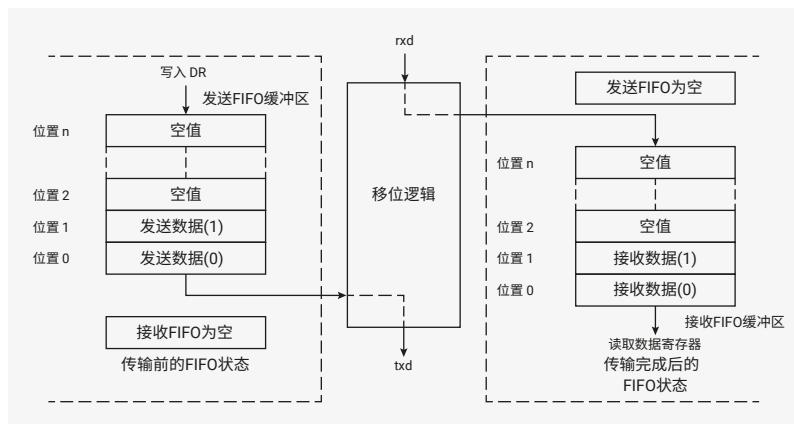
图127。 SPI串行
格式 连续
传输 (SCPH = 1)



DW_apb_ssi支持四种传输模式以执行SPI串行事务。对于发送和接收传输（控制寄存器0中传输模式字段(9:8) = 00b），从DW_apb_ssi发出的数据写入发送FIFO，从外部串行设备接收的数据存入接收FIFO。

图128展示了串行传输开始前及传输完成时的FIFO状态。此示例中，DW_apb_ssi通过连续传输方式向外部串行设备发送了两个数据字。外部串行设备亦会回复两个数据字，用于DW_apb_ssi。

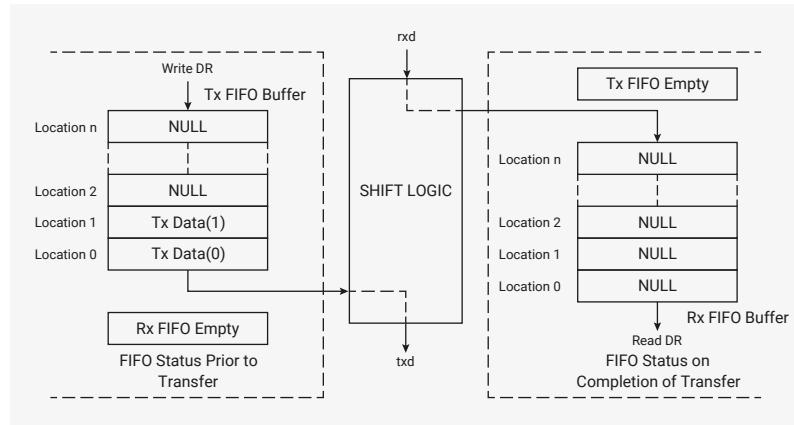
图128。 发送
与接收SPI及SSP传
输的FIFO状态



对于仅发送传输（控制寄存器0的传输模式字段 (9:8) = 01b），从DW_apb_ssi发送到外部串行设备的数据写入发送FIFO。由于从外部串行设备接收的数据被视为无效，故不存储于DW_apb_ssi的接收FIFO中。

[Figure 129](#) shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are transmitted from the DW_apb_ssi to the external serial device in a continuous transfer.

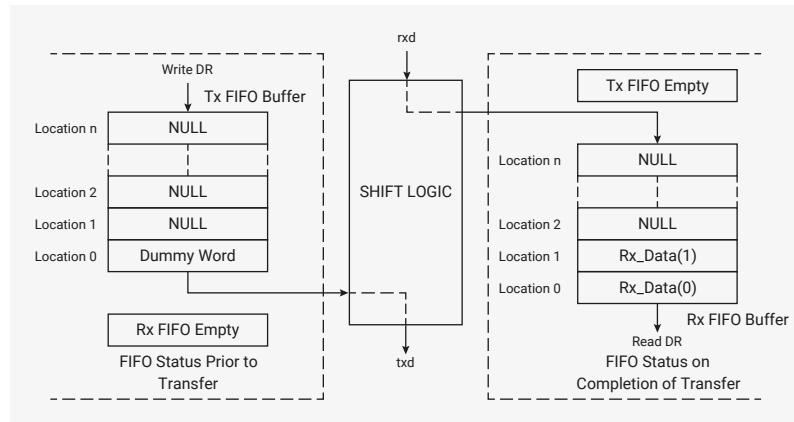
Figure 129. FIFO Status for Transmit Only SPI and SSP Transfers



For receive only transfers (transfer mode field (9:8) of the Control Register 0 = [10b](#)), data transmitted from the DW_apb_ssi to the external serial device is invalid, so a single dummy word is written into the transmit FIFO to begin the serial transfer. The txd output from the DW_apb_ssi is held at a constant logic level for the duration of the serial transfer. Data received from the external serial device into the DW_apb_ssi is pushed into the receive FIFO.

[Figure 130](#) shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, two data words are received by the DW_apb_ssi from the external serial device in a continuous transfer.

Figure 130. FIFO Status for Receive Only SPI and SSP Transfers

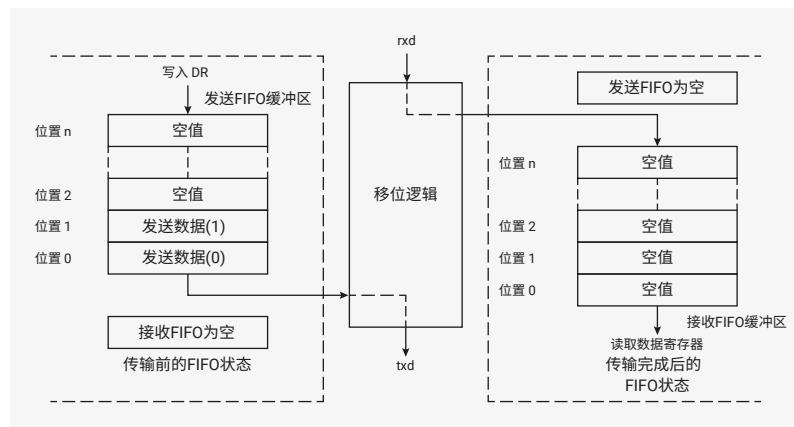


For eeprom_read transfers (transfer mode field [9:8] of the Control Register 0 = [11b](#)), opcode and/or EEPROM address are written into the transmit FIFO. During transmission of these control frames, received data is not captured by the DW_apb_ssi master. After the control frames have been transmitted, receive data from the EEPROM is stored in the receive FIFO.

[Figure 131](#) shows the FIFO levels prior to the beginning of a serial transfer and the FIFO levels on completion of the transfer. In this example, one opcode and an upper and lower address are transmitted to the EEPROM, and eight data frames are read from the EEPROM and stored in the receive FIFO of the DW_apb_ssi master.

图129展示了串行传输开始前及传输完成时的FIFO水位。在此示例中，两个数据字作为连续传输自DW_apb_ssi发送至外部串行设备。

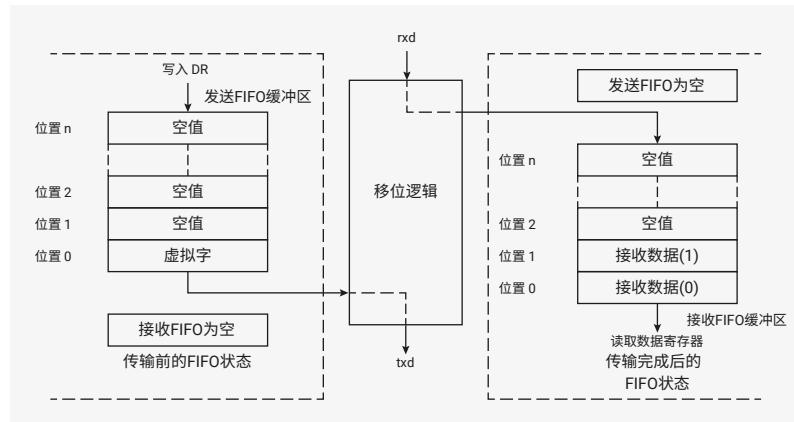
图129。仅针对发送的SPI和SSP传输的FIFO状态



对于仅接收传输（控制寄存器0的传输模式字段 $(9:8) = 10b$ ），从DW_apb_ssi发送至外部串行设备的数据无效，故写入单个伪字至发送FIFO以启动串行传输。DW_apb_ssi的txd输出在整个串行传输过程中保持恒定逻辑电平。来自外部串行设备的数据进入DW_apb_ssi后被推入接收FIFO。

图130展示了串行传输开始前以及传输完成时的 FIFO 水平。在此示例中，DW_apb_ssi 在连续传输过程中从外部串行设备接收了两个数据字。

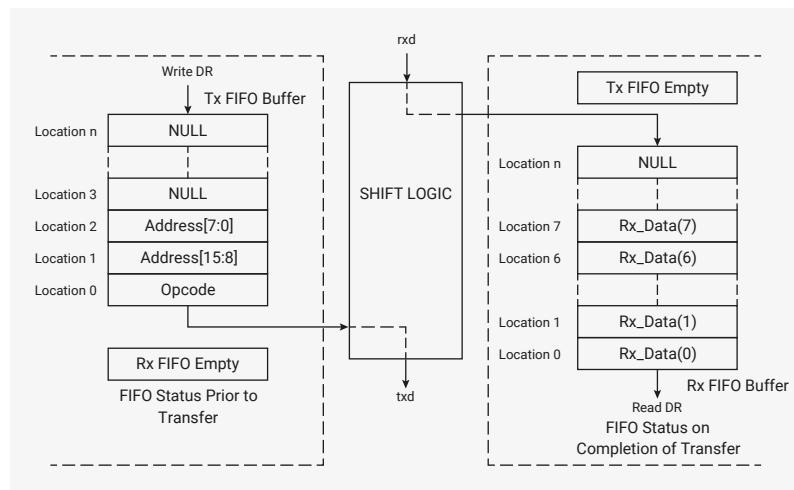
图130. FIFO接收状态
仅限 SPI 和 SSP 传输



对于 eeprom_read 传输（控制寄存器0中传输模式字段[9:8]为 11b），操作码和/或 EEPROM 地址写入发送 FIFO。在传输这些控制帧期间，DW_apb_ssi 主控不会捕获接收的数据。控制帧传输完成后，来自 EEPROM 的接收数据存储于接收 FIFO 中。

图131展示了串行传输开始前以及传输完成时的 FIFO 水平。在本例中，发送一个操作码及高位和低位地址到EEPROM，同时从EEPROM读取八个数据帧，并存储于DW_apb_ssi主控的接收FIFO中。

Figure 131. FIFO Status for EEPROM Read Transfer Mode

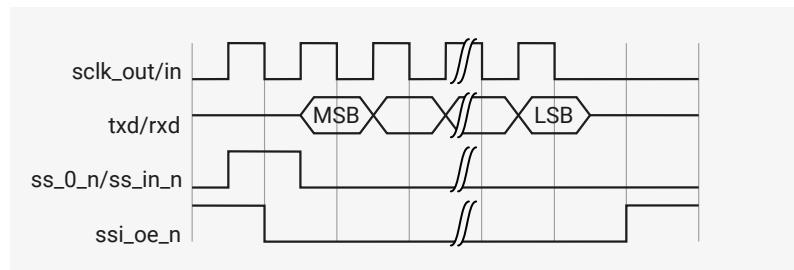


4.10.10.2. Texas Instruments Synchronous Serial Protocol (SSP)

Data transfers begin by asserting the frame indicator line (`ss_0_n/ss_in_n`) for one serial clock period. Data to be transmitted are driven onto the `txd` line one serial clock cycle later; similarly data from the slave are driven onto the `rxd` line. Data are propagated on the rising edge of the serial clock (`sclk_out/sclk_in`) and captured on the falling edge. The length of the data frame ranges from four to 32 bits.

[Figure 132](#) shows the timing diagram for a single SSP serial transfer.

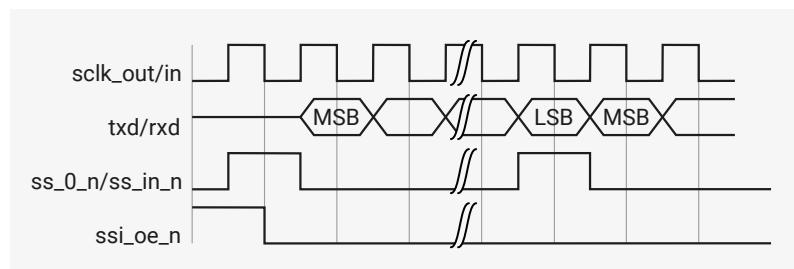
Figure 132. SSP Serial Format



Continuous data frames are transferred in the same way as single data frames. The frame indicator is asserted for one clock period during the same cycle as the LSB from the current transfer, indicating that another data frame follows.

[Figure 133](#) shows the timing for a continuous SSP transfer.

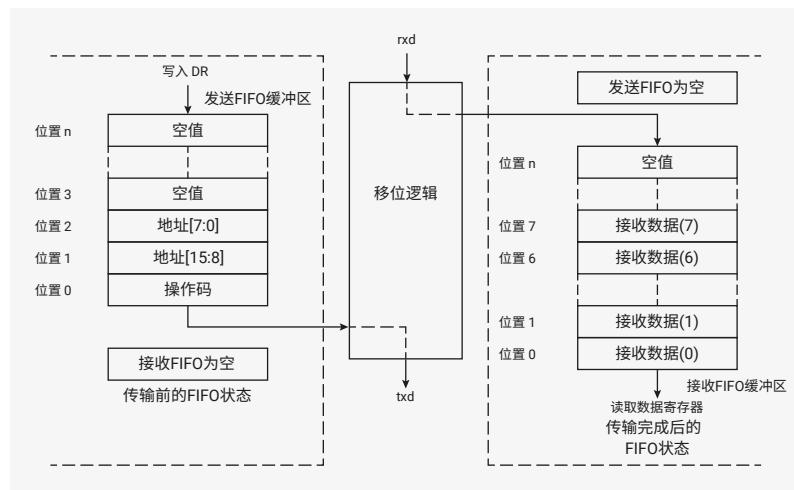
Figure 133. SSP Serial Format Continuous Transfer



4.10.10.3. National Semiconductor Microwire

Data transmission begins with the falling edge of the slave-select signal (`ss_0_n`). One-half serial clock (`sclk_out`) period later, the first bit of the control is sent out on the `txd` line. The length of the control word can be in the range 1 to 16 bits and is set by writing bit field CFS (bits 15:12) in `CTRLR0`. The remainder of the control word is transmitted (propagated on the falling edge of `sclk_out`) by the `DW_apb_ssi` serial master. During this transmission, no data are present (high impedance) on the serial master's `rxd` line.

图131。EEPROM
M读取传输模式
下的FIFO状态

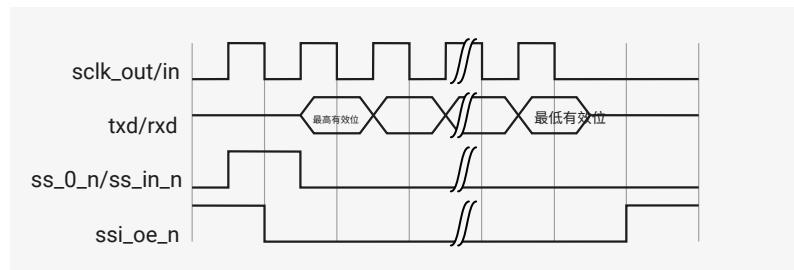


4.10.10.2. 德州仪器同步串行协议 (SSP)

数据传输始于将帧指示线 (ss_0_n/ss_in_n) 断言一个串行时钟周期。待传输的数据在一个串行时钟周期后驱动至txd线；从设备的数据同样驱动至rxd线。数据于串行时钟 (sclk_out/sclk_in) 上升沿传播，并于下降沿采样。数据帧长度范围为4至32位。

图132展示了单个SSP串行传输的时序图。

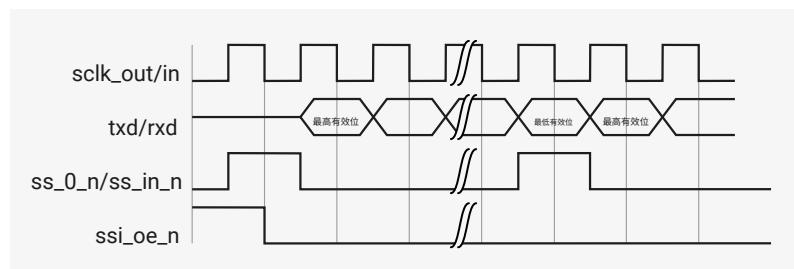
图132. SSP串行
格式



连续数据帧的传输方式与单个数据帧完全相同。帧指示信号在当前传输LSB的同一时钟周期内被确认一个时钟周期，表示后续还有数据帧。

图133展示了连续SSP传输的时序。

图133。SSP串行
格式连续
传输



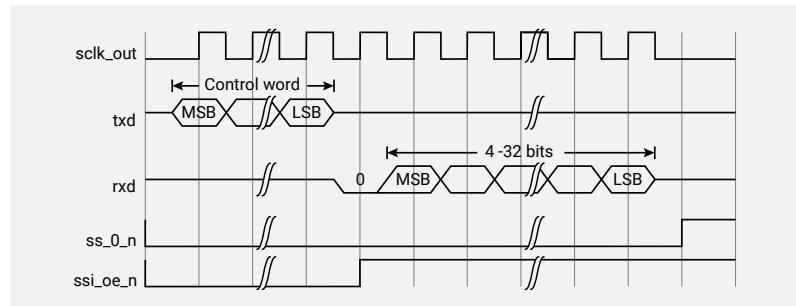
4.10.10.3. 国家半导体Microwire协议

数据传输从从机选择信号 (ss_0_n) 下降沿开始。半个串行时钟周期 (sclk_out) 后，控制字的第一位通过txd线发送。控制字长度范围为1至16位，通过写入CTRLR0寄存器中位域CFS（第15至12位）进行设置。其余控制字由DW_apb_ssi串行主机在sclk_out下降沿传输。在此传输期间，串行主机的rxd线处于无数据（高阻抗）状态。

The direction of the data word is controlled by the MDD bit field (bit 1) in the Microwire Control Register (MWCR). When MDD=0, this indicates that the DW_apb_ssi serial master receives data from the external serial slave. One clock cycle after the LSB of the control word is transmitted, the slave peripheral responds with a dummy 0 bit, followed by the data frame, which can be four to 32 bits in length. Data are propagated on the falling edge of the serial clock and captured on the rising edge.

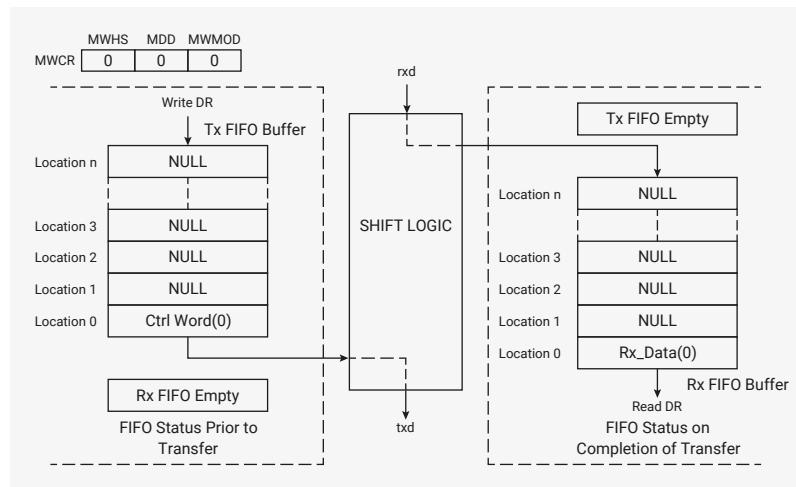
The slave-select signal is held active-low during the transfer and is de-asserted one-half clock cycle later, after the data are transferred. [Figure 134](#) shows the timing diagram for a single DW_apb_ssi serial master read from an external serial slave.

[Figure 134. Single DW_apb_ssi Master Microwire Serial Transfer \(MDD=0\)](#)



[Figure 135](#) shows how the data and control frames are structured in the transmit FIFO prior to the transfer; the value programmed into the MWCR register is also shown.

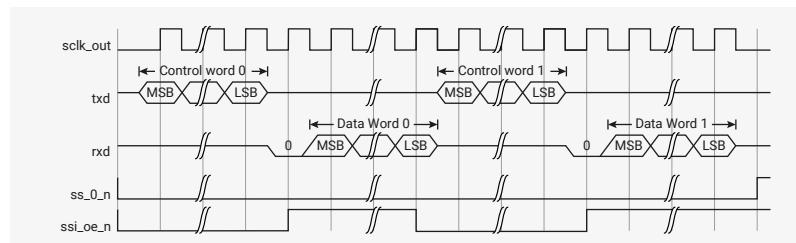
[Figure 135. FIFO Status for Single Microwire Transfer \(receiving data frame\)](#)



Continuous transfers for the Microwire protocol can be sequential or nonsequential, and are controlled by the MWMOD bit field (bit 0) in the MWCR register.

Nonsequential continuous transfers occur as illustrated in [Figure 136](#), with the control word for the next transfer following immediately after the LSB of the current data word.

[Figure 136. Continuous Nonsequential Microwire Transfer \(receiving data frame\)](#)



The only modification needed to perform a continuous nonsequential transfer is to write more control words into the transmit FIFO buffer; this is illustrated in [Figure 137](#). In this example, two data words are read from the external serial-slave device.

数据字的方向由Microwire控制寄存器（MWCR）中的MDD位字段（第1位）控制。当MDD=0时，表示DW_apb_ssi串行主机从外部串行从机接收数据。控制字最低有效位传输完成后一个时钟周期，从机外设响应一个虚拟0位，随后是长度为4至32位的数据帧。数据在串行时钟的下降沿传输，并于上升沿采样。

在传输过程中，从机选择信号保持低电平有效，数据传输完成后半个时钟周期解除使能。图134显示了单个DW_apb_ssi串行主机从外部串行从机读取的时序图。

图134。单通道
DW_apb_ssi主控
Microwire串行
传输（MDD=0）

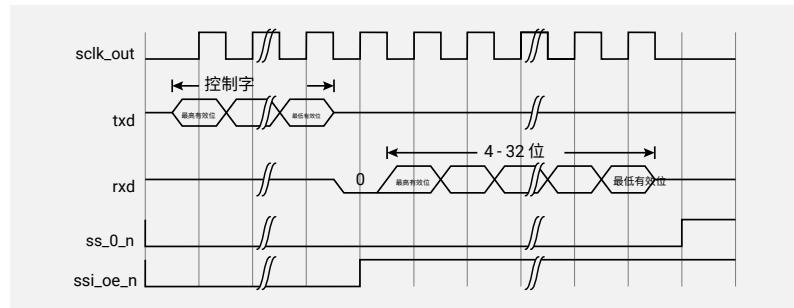
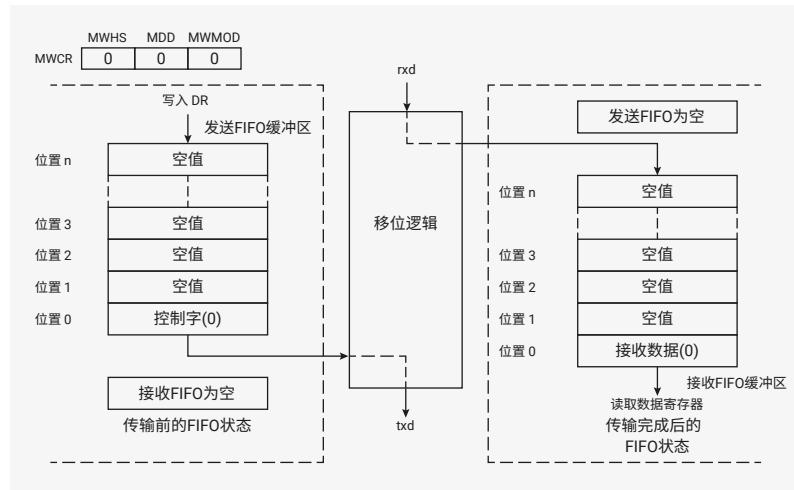


图134展示了单通道DW_apb_ssi主控Microwire串行传输（MDD=0）的时序图。

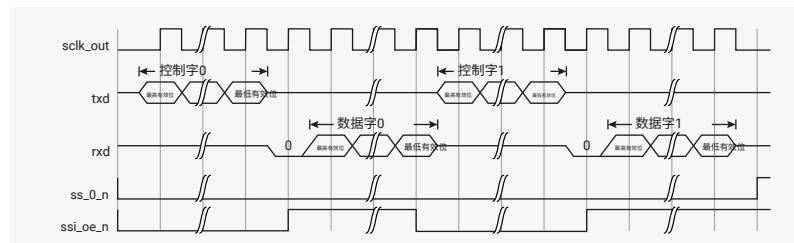
图135。单通道
Microwire传
输的FIFO状态（
接收数据帧）



Microwire协议的连续传输可以是顺序或非顺序的，由MWCR寄存器内的MWMOD位（第0位）控制。

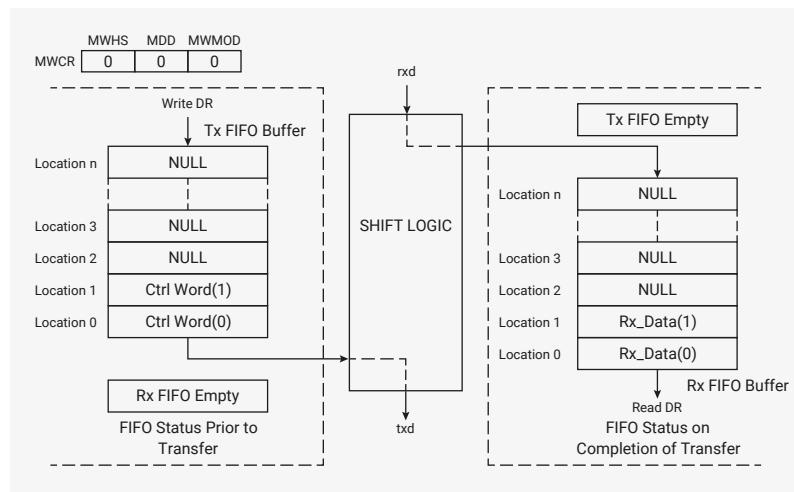
非顺序连续传输如图136所示，下一次传输的控制字紧贴当前数据字的最低有效位之后。

图136。
连续
非连续
Microwire传
输（接收数据帧）



执行连续非连续传输所需的唯一修改是向发送 FIFO 缓冲区写入更多控制字；这在图 137 中有所体现。在本例中，从外部串行从设备读取了两个数据字。

Figure 137. FIFO Status for Nonsequential Microwire Transfer (receiving data frame)



During sequential continuous transfers, only one control word is transmitted from the DW_apb_ssi master. The transfer is started in the same manner as with nonsequential read operations, but the cycle is continued to read further data. The slave device automatically increments its address pointer to the next location and continues to provide data from that location. Any number of locations can be read in this manner; the DW_apb_ssi master terminates the transfer when the number of words received is equal to the value in the CTRL1 register plus one.

The timing diagram in [Figure 138](#) and example in [Figure 139](#) show a continuous sequential read of two data frames from the external slave device.

Figure 138.
Continuous Sequential
Microwire Transfer
(receiving data frame)

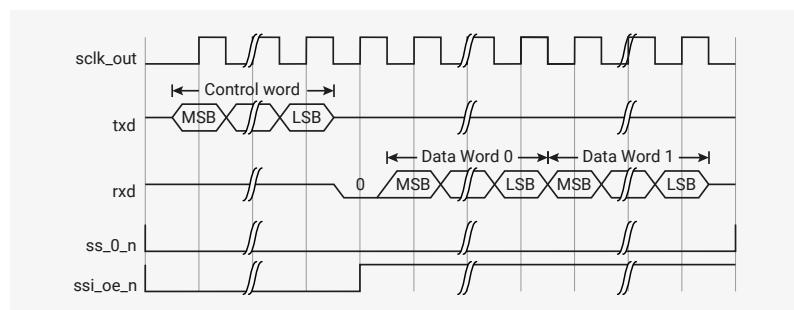
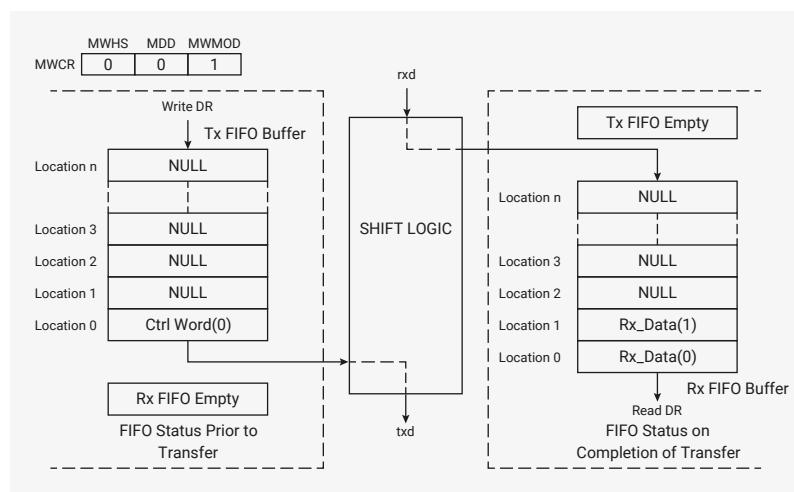


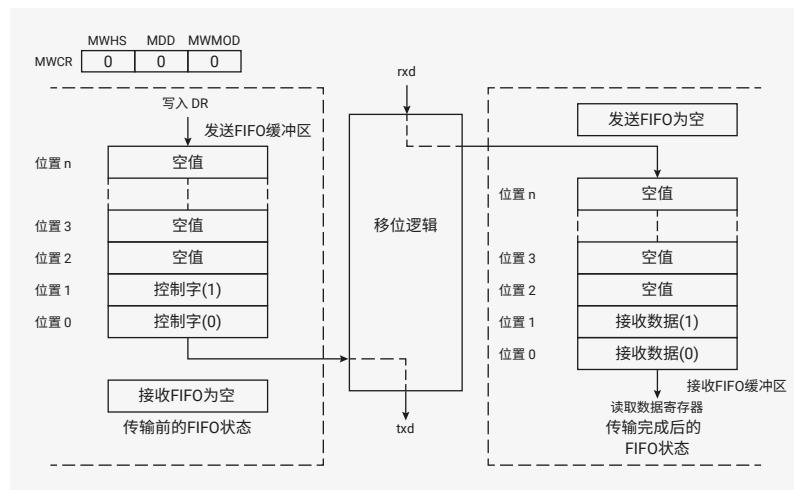
Figure 139. FIFO Status for Sequential Microwire Transfer (receiving data frame)



When MDD = 1, this indicates that the DW_apb_ssi serial master transmits data to the external serial slave. Immediately after the LSB of the control word is transmitted, the DW_apb_ssi master begins transmitting the data frame to the slave peripheral.

[Figure 140](#) shows the timing diagram for a single DW_apb_ssi serial master write to an external serial slave.

图 137。非连续 Microwire 传输（接收数据帧）时的 FIFO 状态



在顺序连续传输过程中，仅由 DW_apb_ssi 主控发送一个控制字。传输的启动方式与非连续读取操作相同，但该周期将继续以读取更多数据。

从设备自动将地址指针递增至下一个位置，并继续从该位置提供数据。可通过此方式读取任意数量的位置；当接收的数据字数达到 CTRLR1 寄存器中数值加一时，DW_apb_ssi 主控终止传输。

图 138 的时序图及图 139 的示例展示了从外部从设备连续顺序读取两个数据帧的过程。

图138。
连续顺序
Microwire传输
(接收数据帧)

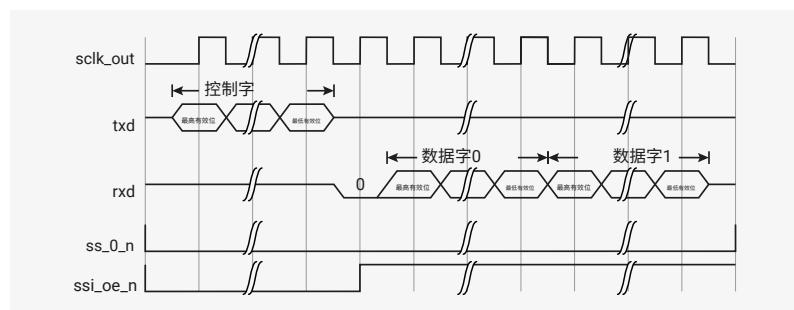
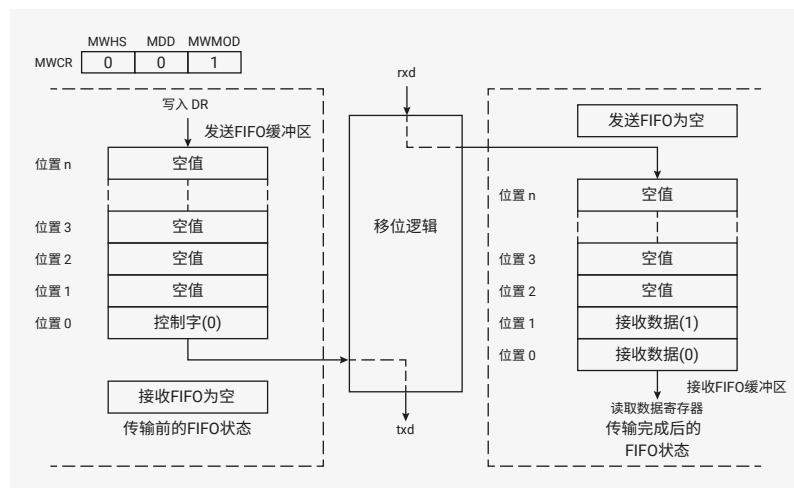


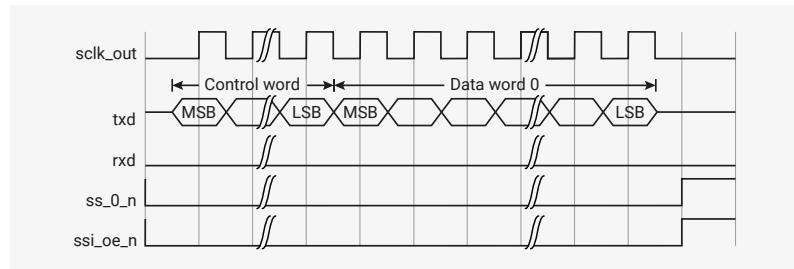
图139。顺序
Microwire传输的FIFO状态 (接收数
据帧)



当MDD=1时，表示DW_apb_ssi串行主设备向外部串行从设备发送数据。在控制字最低有效位传输完成后，DW_apb_ssi 主设备开始向从属外设传输数据帧。

图140显示了单个DW_apb_ssi串行主设备写入外部串行从设备的时序图。

Figure 140. Single Microwire Transfer (transmitting data frame)

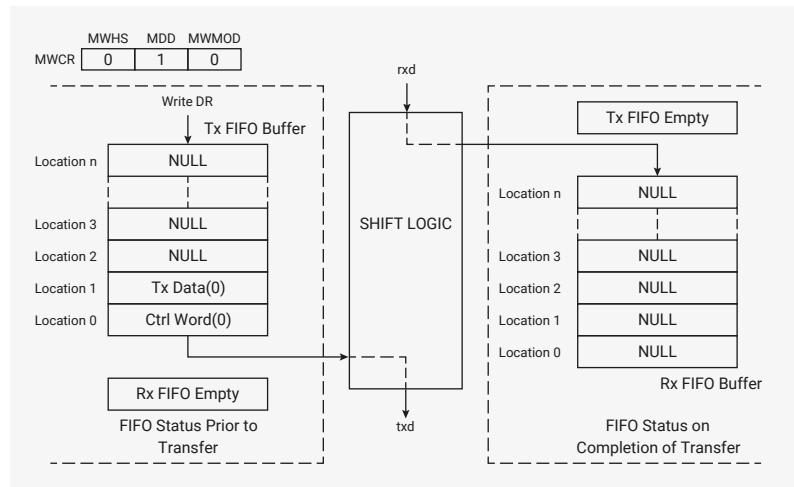


NOTE

The DW_apb_ssi does not support continuous sequential Microwire writes, where MDD = 1 and MWMOD = 1.

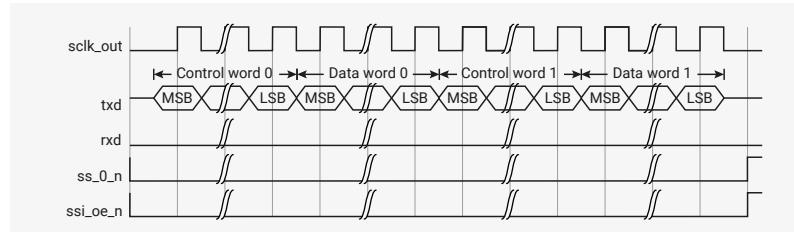
Figure 141 shows how the data and control frames are structured in the transmit FIFO prior to the transfer, also shown is the value programmed into the MWCR register.

Figure 141. FIFO Status for Single Microwire Transfer (transmitting data frame)



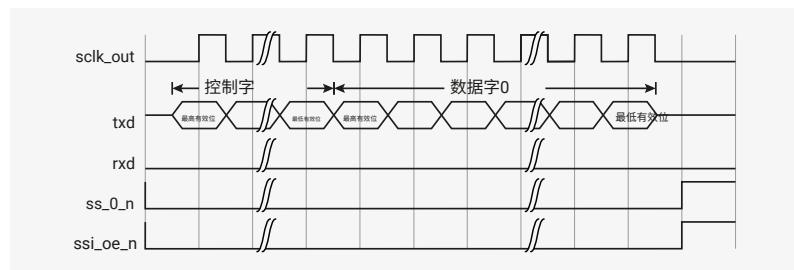
Continuous transfers occur as shown in Figure 142, with the control word for the next transfer following immediately after the LSB of the current data word.

Figure 142. Continuous Microwire Transfer (transmitting data frame)



The only modification you need to make to perform a continuous transfer is to write more control and data words into the transmit FIFO buffer, shown in Figure 143. This example shows two data words are written to the external serial slave device.

图140。单次Microwire传输（发送数据帧）

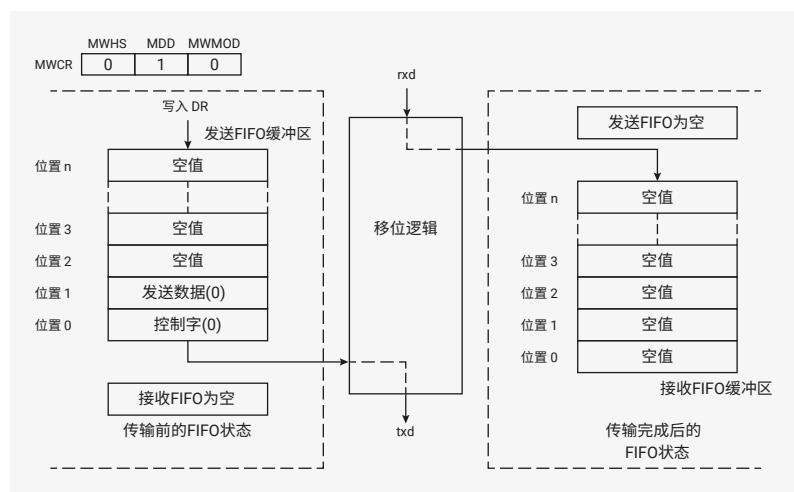


注意

DW_apb_ssi不支持连续顺序Microwire写操作，即MDD=1且MWMOD=1。

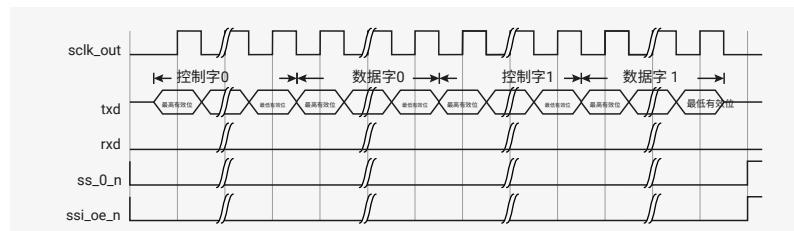
图141展示了传输前发送FIFO中数据帧和控制帧的结构，同时显示了MWCR寄存器中编程的值。

图141。单次Microwire传输的FIFO状态（传输数据帧）



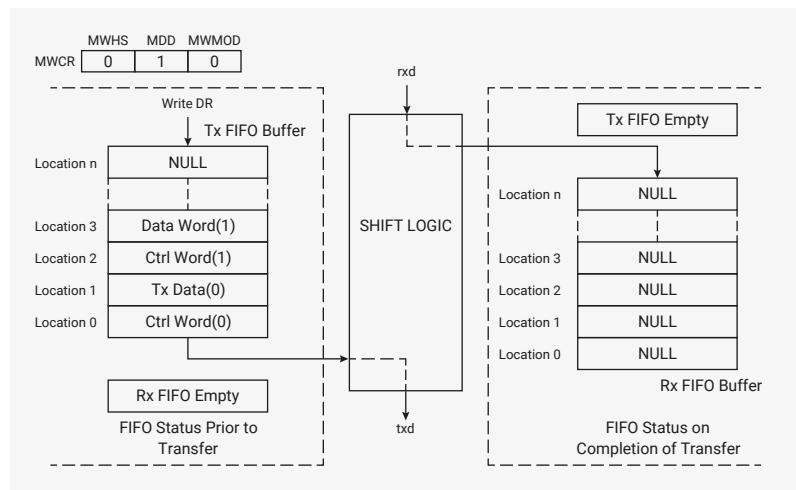
连续传输如图142所示，下一次传输的控制字紧跟当前数据字的最低有效位之后。

图142。
连续 Microwire 传输
(传输数据帧)



执行连续传输的唯一修改是向发送 FIFO 缓冲区写入更多控制字和数据字，见图143。该示例显示向外部串行从设备写入了两个数据字。

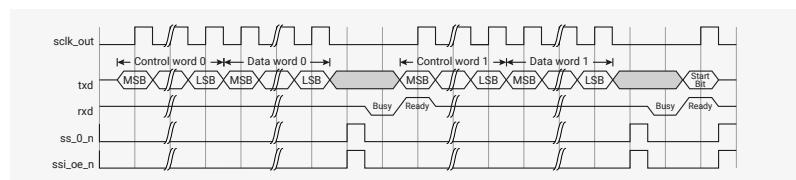
Figure 143. FIFO Status for Continuous Microwire Transfer (transmitting data frame)



The Microwire handshaking interface can also be enabled for DW_apb_ssi master write operations to external serial-slave devices. To enable the handshaking interface, you must write 1 into the MHS bit field (bit 2) on the MWCR register. When MHS is set to 1, the DW_apb_ssi serial master checks for a ready status from the slave device before completing the transfer, or transmitting the next control word for continuous transfers.

Figure 144 shows an example of a continuous Microwire transfer with the handshaking interface enabled.

Figure 144. Continuous Microwire Transfer with Handshaking (transmitting data frame)



After the first data word has been transmitted to the serial-slave device, the DW_apb_ssi master polls the rxd input waiting for a ready status from the slave device. Upon reception of the ready status, the DW_apb_ssi master begins transmission of the next control word. After transmission of the last data frame has completed, the DW_apb_ssi master transmits a start bit to clear the ready status of the slave device before completing the transfer. The FIFO status for this transfer is the same as in [Figure 143](#), except that the MWHS bit field is set (1).

To transmit a control word (not followed by data) to a serial-slave device from the DW_apb_ssi master, there must be only one entry in the transmit FIFO buffer. It is impossible to transmit two control words in a continuous transfer, as the shift logic in the DW_apb_ssi treats the second control word as a data word. When the DW_apb_ssi master transmits only a control word, the MDD bit field (bit 1 of MWCR register) must be set (1).

In the example shown in [Figure 145](#) and in the timing diagram in [Figure 146](#), the handshaking interface is enabled. If the handshaking interface is disabled (MHS=0), the transfer is terminated by the DW_apb_ssi master one sclk_out cycle after the LSB of the control word is captured by the slave device.

Figure 145. FIFO Status for Microwire Control Word Transfer

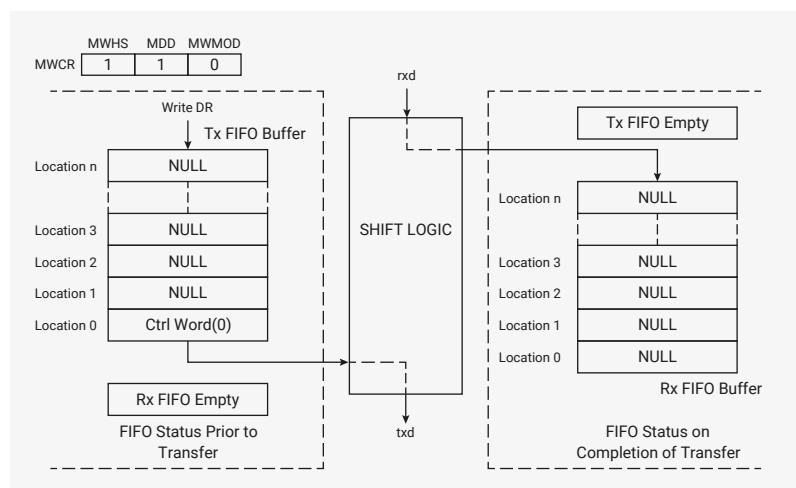
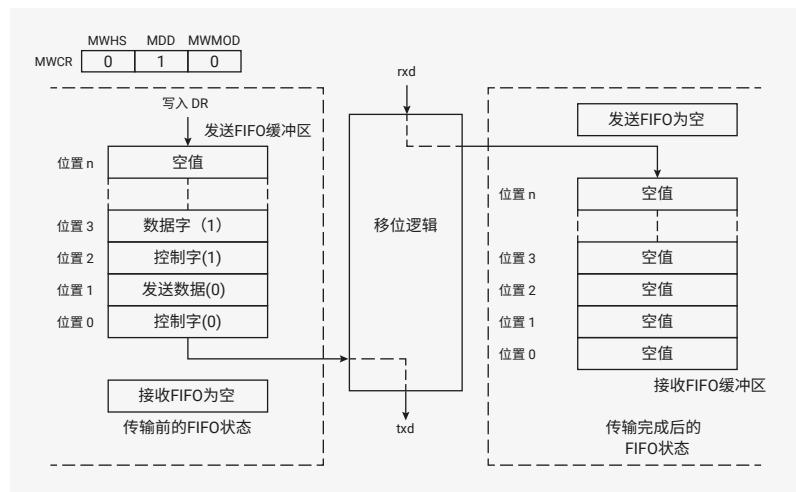


图143。连续
Microwire传输的FIFO状态（数据帧发送）

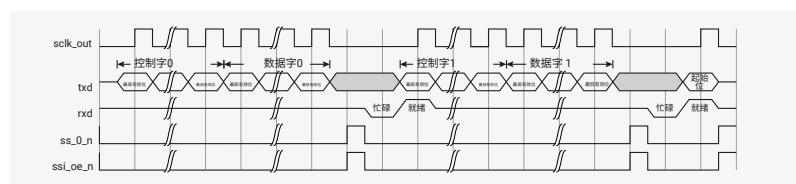


Microwire 握手接口也可用于 DW_apb_ssi 主设备对外部串行从设备的写操作。要启用握手接口，须在 MWCR 寄存器的 MHS 位域（第2位）写入1。

当 MHS 设置为 1 时，DW_apb_ssi 串行主控在完成传输前，或执行连续传输时发送下一个控制字之前，会先检查从设备的就绪状态。

图 144 展示了启用握手接口的连续 Microwire 传输示例。

图 144。
带握手的连续 Micro
wire 传输（
数据帧发送
）



第一个数据字发送至串行从设备后，DW_apb_ssi 主控轮询 rxd 输入端口，等待从设备返回就绪状态。接收到就绪状态后，DW_apb_ssi 主控开始发送下一个控制字。最后一个数据帧发送完成后，DW_apb_ssi 主控发送起始位以清除从设备就绪状态，然后完成传输。该传输的 FIFO 状态与图 143 相同，仅 MWHS 位字段被设置为 1。

要从 DW_apb_ssi 主设备向串行从设备传输控制字（且不跟随数据），传输 FIFO 缓冲区中必须仅包含一条数据。无法在连续传输中发送两个控制字，因为 DW_apb_ssi 的移位逻辑会将第二个控制字识别为数据字。当 DW_apb_ssi 主设备仅传输控制字时，须将 MWCR 寄存器第 1 位的 MDD 位设置为 1。

如图 145 所示示例及图 146 的时序图中，握手接口已启用。若握手接口禁用（MHS=0），则在从设备捕获控制字最低有效位之后的一个 sclk_out 周期内，DW_apb_ssi 主设备将终止传输。

图 145。Micro
wire控制字传输的
FIFO状态

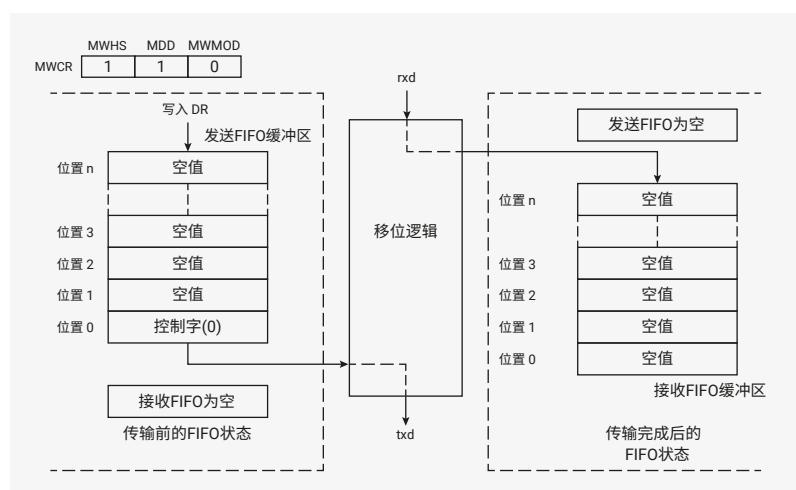
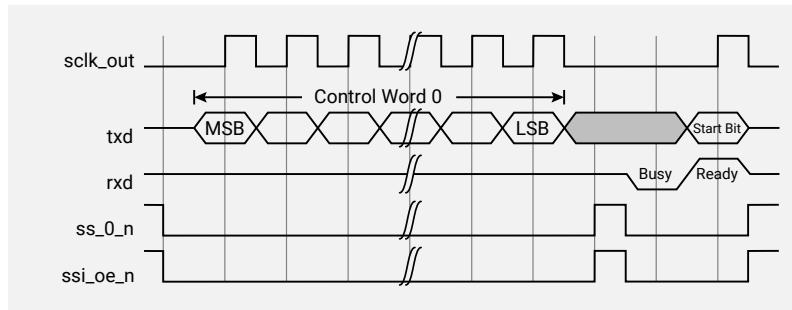


Figure 146. Microwire Control Word



4.10.10.4. Enhanced SPI Modes

DW_apb_ssi supports the dual and quad modes of SPI in RP2040; octal mode is not supported. txd, rxd and ssi_oe_n signals are four bits wide.

Data is shifted out/in on more than one line, increasing the overall throughput. All four combinations of the serial clock's polarity and phase are valid in this mode and work the same as in normal SPI mode. Dual SPI, or Quad SPI modes function similarly except for the width of txd, rxd and ssi_oe_n signals. The mode of operation (write/read) can be selected using the CTRLR0.TMOD field.

4.10.10.4.1. Write Operation in Enhanced SPI Modes

Dual, or Quad, SPI write operations can be divided into three parts:

- Instruction phase
- Address phase
- Data phase

The following register fields are used for a write operation:

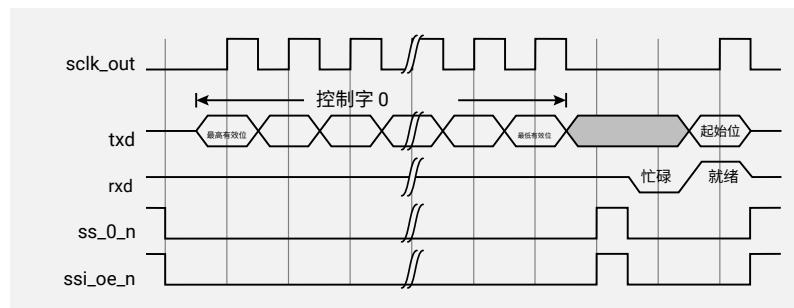
- CTRLR0.SPI_FRF - Specifies the format in which the transmission happens for the frame.
- SPI_CTRLR0 (Control Register 0 register) – Specifies length of instruction, address, and data.
- SPI_CTRLR0.INST_L – Specifies length of an instruction (possible values for an instruction are 0, 4, 8, or 16 bits.)
- SPI_CTRLR0.ADDR_L – Specifies address length (See [Table 579](#) for decode values)
- CTRLR0.DFS or CTRLR0.DFS_32 – Specifies data length.

An instruction takes one FIFO location. An address can take more than one FIFO locations.

Both the instruction and address must be programmed in the data register (DR). DW_apb_ssi will wait until both have been programmed to start the write operation.

The instruction, address and data can be programmed to send in dual/quad mode, which can be selected from the SPI_CTRLR0.TRANS_TYPE and CTRLR0.SPI_FRF fields.

图146. Microwire
控制字



4.10.10.4. 增强型 SPI 模式

DW_apb_ssi 支持 RP2040 中 SPI 的双模和四模；不支持八模。txd、rxd 及 ssi_oe_n 信号宽度均为四位。

数据通过多条线路进行移出与移入，从而提升整体吞吐量。该模式支持串行时钟极性和相位的全部四种组合，且其行为与普通 SPI 模式相同。双 SPI 与四 SPI 模式功能相似，唯一差异在于 txd、rxd 及 ssi_oe_n 信号的宽度。操作模式（写入/读取）可通过 CTRLR0.TMOD 字段进行选择。

4.10.10.4.1. 增强型 SPI 模式中的写操作

双 SPI 或四 SPI 写操作可分为三个阶段：

- 指令阶段
- 地址阶段
- 数据阶段

以下寄存器字段用于写操作：

- CTRLR0.SPI_FRF — 指定帧传输的格式。
- SPI_CTRLR0 (控制寄存器0) — 指定指令、地址和数据的长度。
- SPI_CTRLR0.INST_L — 指定指令长度（可选值为0、4、8或16位）。
- SPI_CTRLR0.ADDR_L — 指定地址长度（详见表579的解释值）。
- CTRLR0.DFS 或 CTRLR0.DFS_32 — 指定数据长度。

一条指令占用一个FIFO位置。一个地址可能占用多个FIFO位置。

指令和地址必须均编入数据寄存器（DR）。DW_apb_ssi将等待指令和地址均编入后，才开始写操作。

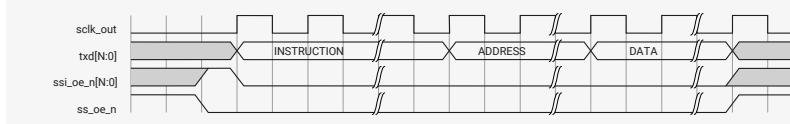
指令、地址及数据可在双线/四线模式下发送，该模式由SPI_CTRLR0.TRANS_TYPE和CTRLR0.SPI_FRF字段选择。

NOTE

- If CTRLR0.SPI_FRF is selected to be "Standard SPI Format", everything is sent in Standard SPI mode and SPI_CTRLR0.TRANS_TYPE field is ignored.
- CTRLR0.SPI_FRF is only applicable if CTRLR0.FRF is programmed to **00b**.

[Figure 147](#) shows a typical write operation in Dual, or Quad, SPI Mode. The value of N will be: 7 if SSI_SPI_MODE is set to 3, 3 if SSI_SPI_MODE is set to 2, and 1 if SSI_SPI_MODE is set to 1. For 1-write operation, the instruction and address are sent only once followed by data frames programmed in DR until the transmit FIFO becomes empty.

Figure 147. Typical Write Operation Dual/Quad SPI Mode

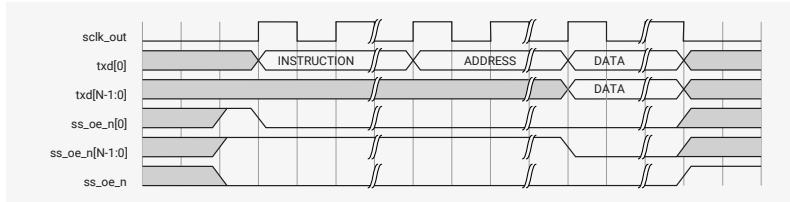


To initiate a Dual/Quad write operation, CTRLR0.SPI_FRF must be set to 01/10/11, respectively. This will set the transfer type, and for each write command, data will be transferred in the format specified in CTRLR0.SPI_FRF field.

Case A: Instruction and address both transmitted in standard SPI format

For this, SPI_CTRLR0.TRANS_TYPE field must be set to **00b**. [Figure 148](#) shows the timing diagram when both instruction and address are transmitted in standard SPI format. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

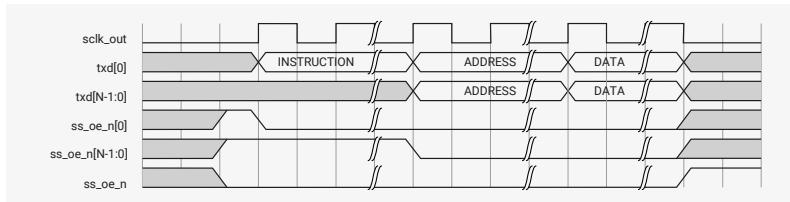
Figure 148. Instruction and Address Transmitted in Standard SPI Format



Case B: Instruction transmitted in standard and address transmitted in Enhanced SPI format

For this, SPI_CTRLR0.TRANS_TYPE field must be set to one. [Figure 149](#) shows the timing diagram when an instruction is transmitted in standard format and address is transmitted in dual SPI format specified in the CTRLR0.SPI_FRF field. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

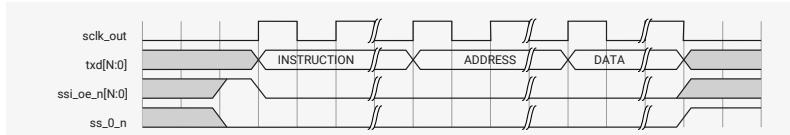
Figure 149. Instruction Transmitted in Standard and Address Transmitted in Enhanced SPI Format



Case C: Instruction and Address both transmitted in Enhanced SPI format

For this, SPI_CTRLR0.TRANS_TYPE field must be set to **10b**. [Figure 150](#) shows the timing diagram in which instruction and address are transmitted in SPI format specified in the CTRLR0.SPI_FRF field. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

Figure 150. Instruction and Address Both Transmitted in Enhanced SPI Format



Case D: Instruction only transfer in enhanced SPI format

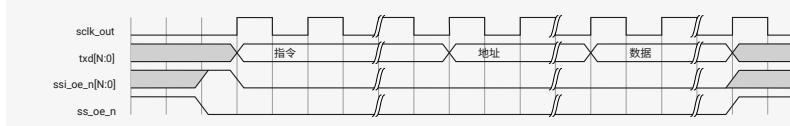
For this, SPI_CTRLR0.TRANS_TYPE field must be set to **10b**. [Figure 151](#) shows the timing diagram for such a transfer. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if

注意

- 若CTRLR0.SPI_FRF选择为“标准SPI格式”，则所有数据均以标准SPI模式发送，SPI_CTRLR0.TRANS_TYPE字段将被忽略。
- CTRLR0.SPI_FRF仅在CTRLR0.FRF被编程为`00b`时适用。

图147展示了在双线或四线SPI模式下的典型写操作。当SSI_SPI_MODE设置为3时，N值为7；设置为2时，N值为3；设置为1时，N值为1。对于一次写操作，指令和地址仅发送一次，随后发送DR中编程的数据帧，直到发送FIFO为空。

图147。典型写操作
双线/四线SPI模式

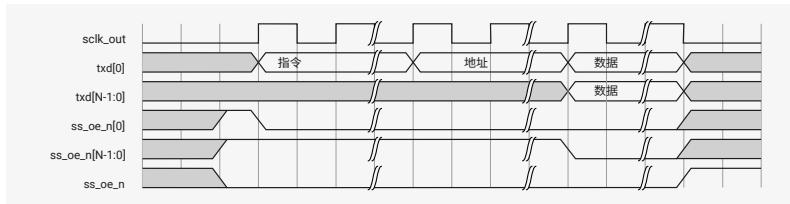


要启动双线/四线写操作，CTRLR0.SPI_FRF须分别设置为01、10或11。此操作将设置传输类型，且每条写命令的数据均按照CTRLR0.SPI_FRF字段指定的格式进行传输。

情况A：指令和地址均以标准SPI格式传输

为此，SPI_CTRLR0.TRANS_TYPE字段须设为`00b`。图148显示了指令和地址均以标准SPI格式传输时的时序图。当CTRLR0.SPI_FRF设为11b时，N值为7；设为10b时，N值为3；设为01b时，N值为1。

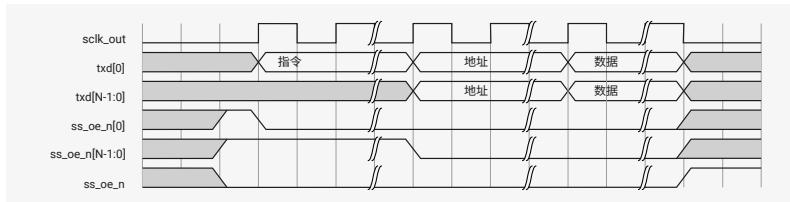
图148。指令和地址以标准SPI格式传输



情况B：指令以标准格式传输，地址以增强SPI格式传输

为此，必须将SPI_CTRLR0.TRANS_TYPE字段设置为1。图149展示了在指令以标准格式传输且地址以增强SPI格式传输时的时序图。当CTRLR0.SPI_FRF设置为`11b`时，N的值为7；设置为`10b`时，N的值为3；设置为`01b`时，N的值为1。

图149。指令以标准格式传输，地址以增强SPI格式传输

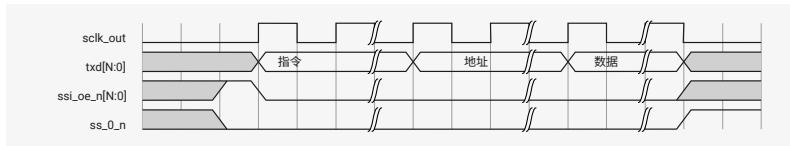


情况C：指令和地址均以增强SPI格式传输

为此，必须将SPI_CTRLR0.TRANS_TYPE字段设置为`10b`。图150展示了指令和地址以CTRLR0.SPI_FRF字段指定之SPI格式传输的时序图。N的值为：

当CTRLR0.SPI_FRF设置为`11b`时，N的值为7；设置为`10b`时为3；设置为`01b`时为1。

图150。增强SPI格式中同时传输指令和地址

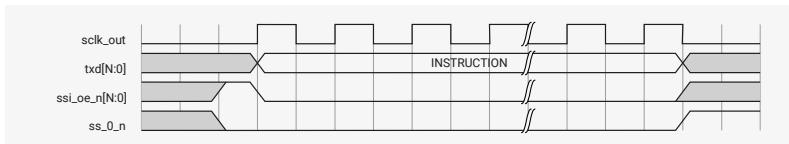


情况D：增强SPI格式中仅传输指令

为此，SPI_CTRLR0.TRANS_TYPE字段必须设置为`10b`。图151显示了该传输的时序图。N的值为：当CTRLR0.SPI_FRF设置为`11b`时为7；设置为`10b`时为3；且当

CTRLR0.SPI_FRF is set to **01b**.

Figure 151. Instruction only transfer in enhanced SPI Format



4.10.10.4.2. Read Operation in Enhanced SPI Modes

A Dual, or Quad, SPI read operation can be divided into four phases:

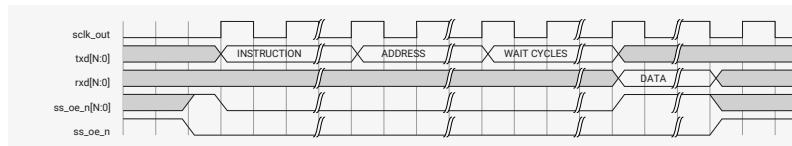
- Instruction phase
- Address phase
- Wait cycles
- Data phase

Wait Cycles can be programmed using SPI_CTRLR0.WAIT_CYCLES field. The value programmed into SPI_CTRLR0.WAIT_CYCLES is mapped directly to sclk_out times. For example, WAIT_CYCLES=0 indicates no Wait, WAIT_CYCLES=1, indicates one wait cycle and so on. The wait cycles are introduced for target slave to change their mode from input to output and the wait cycles can vary for different devices.

For a READ operation, DW_apb_ssi sends instruction and control data once and waits until it receives NDF (CTRLR1 register) number of data frames and then de-asserts slave select signal.

Figure 152 shows a typical read operation in dual quad SPI mode. The value of N will be: 3 if SSI_SPI_MODE is set to Quad mode, and 1 if SSI_SPI_MODE is set to Dual mode.

Figure 152. Typical Read Operation in Enhanced SPI Mode



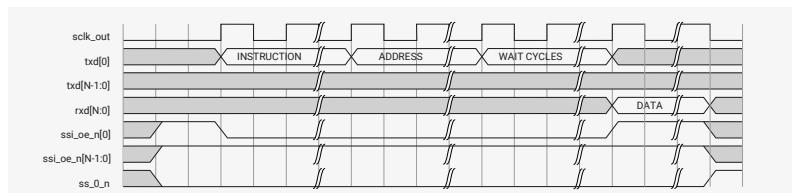
To initiate a dual/quad read operation, CTRLR0.SPI_FRF must be set to 01/10/11 respectively. This will set the transfer type, now for each read command data will be transferred in the format specified in CTRLR0.SPI_FRF field.

Following are the possible cases of write operation in enhanced SPI modes:

Case A: Instruction and address both transmitted in standard SPI format

For this, SPI_CTRLR0.TRANS_TYPE field should be set to **00b**. Figure 153 shows the timing diagram when both instruction and address are transferred in standard SPI format. The figure also shows WAIT cycles after address, which can be programmed in the SPI_CTRLR0.WAIT_CYCLES field. The value of N will be 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

Figure 153. Instruction and Address Transmitted in Standard SPI Format

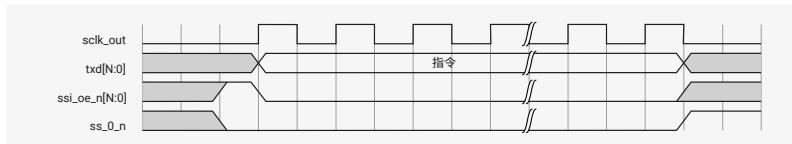


Case B: Instruction transmitted in standard and address transmitted in dual SPI format

For this, SPI_CTRLR0.TRANS_TYPE field should be set to **01b**. Figure 154 shows the timing diagram in which instruction is transmitted in standard format and address is transmitted in dual SPI format. The value of N will be 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

CTRLR0.SPI_FRF 设置为 **01b** 时为 1。

图 151。增强 SPI 格式中仅传输指令



4.10.10.4.2. 增强 SPI 模式下的读取操作 双线或四线 SPI

读取操作可分为四个阶段：

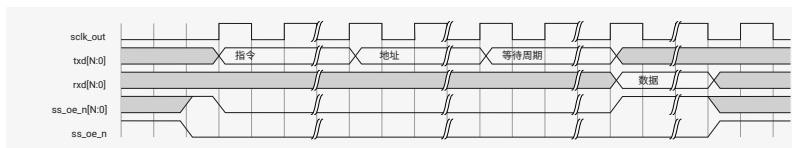
- 指令阶段
- 地址阶段
- 等待周期
- 数据阶段

等待周期可通过 SPI_CTRLR0.WAIT_CYCLES 字段进行编程。编程进 SPI_CTRLR0.WAIT_CYCLES 的值直接对应于 sclk_out 的周期数。例如，WAIT_CYCLES=0 表示无等待，WAIT_CYCLES=1 表示等待一个周期，依此类推。等待周期用于使目标从机从输入模式切换至输出模式，不同设备的等待周期可能不同。

对于读取操作，DW_apb_ssi 一次性发送指令和控制数据，等待直到接收 NDF（CTRLR1 寄存器）指定数量的数据帧，随后取消从机选择信号。

图152展示了双四线 SPI 模式下的典型读取操作。当 SSI_SPI_MODE 设为四线模式时，N 的值为3；当 SSI_SPI_MODE 设为双线模式时，N 的值为1。

图152。增强SPI模式下的典型读取操作



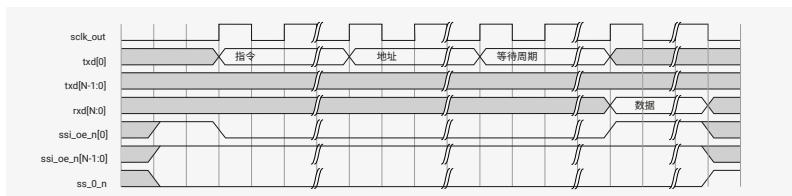
要启动双路/四路读取操作，CTRLR0.SPI_FRF 必须分别设置为 01、10 或 11。此设置确定传输类型，之后每个读取命令的数据将按照 CTRL0.SPI_FRF 字段指定的格式传输。

以下为增强SPI模式下写操作的可能情况：

情况A：指令和地址均以标准SPI格式传输

为此，SPI_CTRLR0.TRANS_TYPE 字段应设置为 **00b**。图153显示了指令和地址均以标准SPI格式传输时的时序图。图中还展示了地址之后的等待周期，可通过 SPI_CTRLR0.WAIT_CYCLES 字段进行编程。当 CTRLR0.SPI_FRF 设置为 **11b** 时，N 的值为 7；设置为 **10b** 时，N 为 3；设置为 **01b** 时，N 为 1。

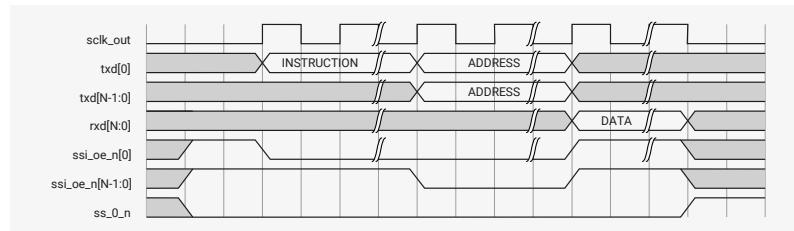
图153。指令和地址以标准SPI格式传输



情况B：指令以标准格式传输，地址以双路SPI格式传输

为此，应将 SPI_CTRLR0.TRANS_TYPE 字段设置为 **01b**。图154显示了指令以标准格式传输，地址以双路SPI格式传输的时序图。当 CTRLR0.SPI_FRF 设置为 **11b** 时，N 的值为 7；设置为 **10b** 时，N 为 3；设置为 **01b** 时，N 为 1。

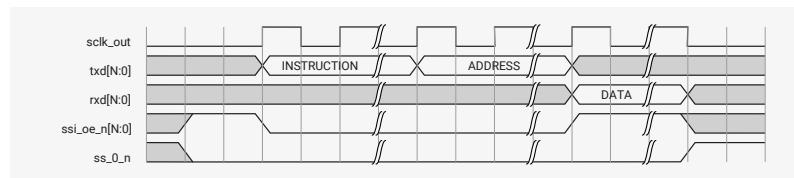
Figure 154. Instruction Transmitted in Standard and Address Transmitted in Enhanced SPI Format



Case C: Instruction and Address both transmitted in Dual SPI format

For this, SPI_CTRLR0.TRANS_TYPE field must be set to **10b**. Figure 155 shows the timing diagram in which both instruction and address are transmitted in dual SPI format. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**.

Figure 155. Instruction and Address Transmitted in Enhanced SPI Format



Case D: No Instruction, No Address READ transfer

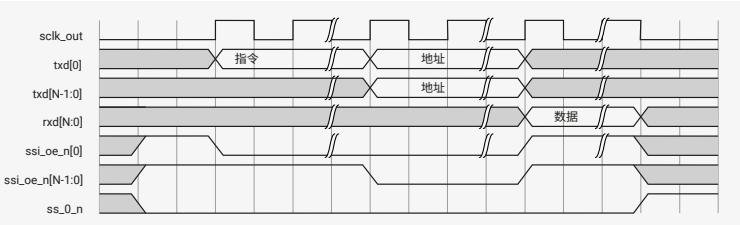
For this, SPI_CTRLR0.ADDR_L and SPI_CTRLR0.INST_L must be set to 0 and SPI_CTRLR0.WAIT_CYCLES must be set to a non-zero value. Table 579 lists the ADDR_L decode value and the respective description for enhanced (Dual/Quad) SPI modes.

Table 579. ADDR_L Decode in Enhanced SPI Mode

ADDR_L Decode Value	Description
0000	0-bit Address Width
0001	4-bit Address Width
0010	8-bit Address Width
0011	12-bit Address Width
0100	16-bit Address Width
0101	20-bit Address Width
0110	24-bit Address Width
0111	28-bit Address Width
1000	32-bit Address Width
1001	36-bit Address Width
1010	40-bit Address Width
1011	44-bit Address Width
1100	48-bit Address Width
1101	52-bit Address Width
1110	56-bit Address Width
1111	60-bit Address Width

Figure 156 shows the timing diagram for such type of transfer. The value of N will be: 7 if CTRLR0.SPI_FRF is set to **11b**, 3 if CTRLR0.SPI_FRF is set to **10b**, and 1 if CTRLR0.SPI_FRF is set to **01b**. To initiate this transfer, the software has to perform a dummy write in the data register (DR), DW_apb_ssi will wait for programmed wait cycles and then fetch the amount of data specified in NDF field.

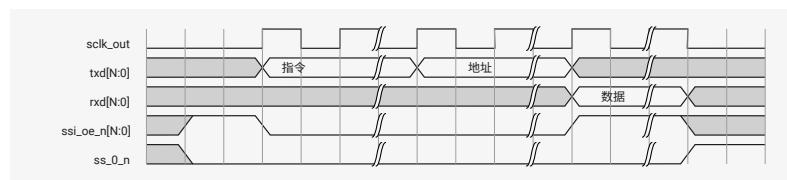
图154。指令以标准格式传输，地址以增强 SPI 格式传输



情况C：指令和地址均以双路SPI格式传输

为此，`SPI_CTRLR0.TRANS_TYPE` 字段必须设置为 `10b`。图155显示了指令和地址均以双路SPI格式传输的时序图。当 `CTRLR0.SPI_FRF` 设置为 `11b` 时，N 的值为 7；设置为 `10b` 时，N 的值为 3；设置为 `01b` 时，N 的值为 1。

图155。指令及地址以增强型SPI格式传输



情况D：无指令、无地址的读操作传输

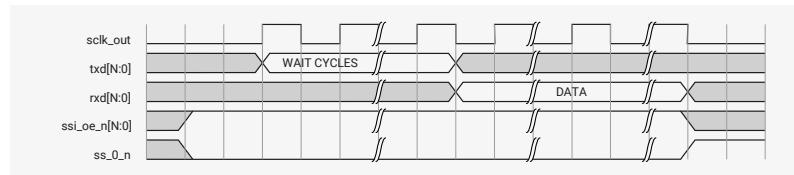
为此，`SPI_CTRLR0.ADDR_L`与`SPI_CTRLR0.INST_L`须设置为0，且`SPI_CTRLR0.WAIT_CYCLES`须设置为非零值。表5-79列出了`ADDR_L`解码值及其对应的增强（双/四）SPI模式说明。

表579. `ADDR_L`
增强模式下的解码
SPI模式

<code>ADDR_L</code> 解码值	描述
0000	0位地址宽度
0001	4位地址宽度
0010	8位地址宽度
0011	12位地址宽度
0100	16位地址宽度
0101	20位地址宽度
0110	24位地址宽度
0111	28位地址宽度
1000	32位地址宽度
1001	36位地址宽度
1010	40位地址宽度
1011	44位地址宽度
1100	48位地址宽度
1101	52位地址宽度
1110	56位地址宽度
1111	60位地址宽度

图156显示了该类型传输的时序图。若 `CTRLR0.SPI_FRF` 设置为 `11b`，则 N 值为 7；若设置为 `10b`，则 N 值为 3；若设置为 `01b`，则 N 值为 1。要启动此传输，软件必须在数据寄存器（DR）中执行虚拟写操作，`DW_apb_ssi` 将等待设定的等待周期，然后读取 NDF 字段指定的数据量。

Figure 156. No
Instruction and No
Address READ
Transfer



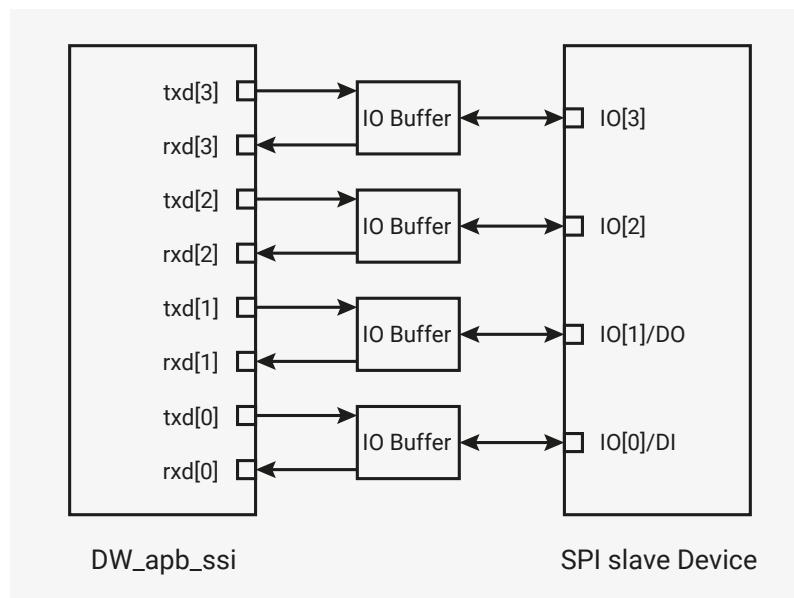
4.10.10.4.3. Advanced I/O Mapping for Enhanced SPI Modes

The Input/Output mapping for enhanced SPI modes (dual, and quad) is hardcoded inside the DW_apb_ssi. The rxd[1] signal will be used to sample incoming data in standard SPI mode of operation.

For other protocols (such as SSP and Microwire), the I/O mapping remains the same. Therefore, it is easy for other protocols to connect with any device that supports Dual/Quad SPI operation because other protocols do not require a MUX logic to exist outside the design.

[Figure 157](#) shows the I/O mapping of DW_apb_ssi in Quad mode with another SPI device that supports the Quad mode. As illustrated in [Figure 157](#), the IO[1] pin is used as DO in standard SPI mode of operation and it is connected to rxd[1] pin, which will be sampling the input in the standard mode of operation.

Figure 157. Advanced
I/O Mapping in Quad
SPI Modes



4.10.10.5. Dual Data-Rate (DDR) Support in SPI Operation

In standard operations, data transfer in SPI modes occur on either the positive or negative edge of the clock. For improved throughput, the dual data-rate transfer can be used for reading or writing to the memories.

The DDR mode supports the following modes of SPI protocol:

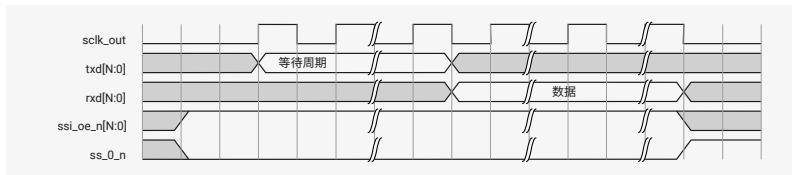
- SCPH=0 & SCPOL=0 (Mode 0)
- SCPH=1 & SCPOL=1 (Mode 3)

DDR commands enable data to be transferred on both edges of clock. Following are the different types of DDR commands:

- Address and data are transmitted (or received in case of data) in DDR format, while instruction is transmitted in standard format.
- Instruction, address, and data are all transmitted or received in DDR format.

The DDR_EN (SPI_CTRLR0[16]) bit is used to determine if the Address and data have to be transferred in DDR mode and INST_DDR_EN (SPI_CTRLR0[17]) bit is used to determine if Instruction must be transferred in DDR format. These bits

图156。无
指令且无
地址读
传输



4.10.10.4.3. 增强SPI模式的高级I/O映射

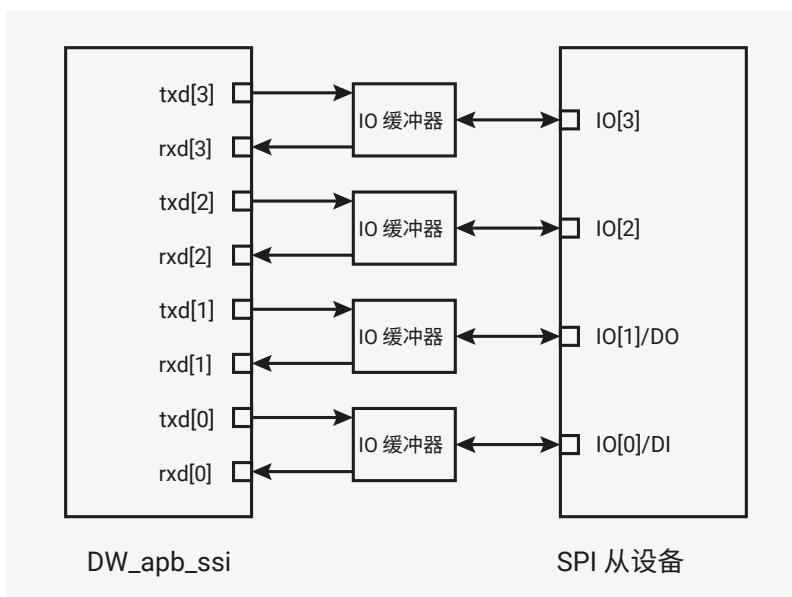
增强SPI模式（双线和四线）的输入输出映射已硬编码于DW_apb_ssi中。在标准SPI工作模式下，rx[1]信号用于采样接收数据。

对于其他协议（如 SSP 和 Microwire），I/O 映射保持不变。因此，其他协议能够轻松连接任何支持双通道/四通道 SPI 操作的设备，因为这些协议不需要设计外部存在多路复用（MUX）逻辑。

[图157展示了 DW_apb_ssi 在四通道模式下与另一支持四通道模式的 SPI 设备的 I/O 映射关系。](#)

如图157所示，IO[1] 引脚在标准 SPI 操作模式中用作 DO，且连接至 rx[1] 引脚，该引脚将在标准操作模式中采样输入信号。

图157. 高级
四路中的I/O映射
SPI模式



4.10.10.5. SPI操作中的双数据率（DDR）支持

在标准操作中，SPI模式下的数据传输发生于时钟的上升沿或下降沿。为提高吞吐量，可使用双数据率传输进行存储器读写。

DDR模式支持以下SPI协议模式：

- SCPH=0 且 SCPOL=0（模式0）
- SCPH=1 且 SCPOL=1（模式3）

DDR命令允许数据在时钟的双边沿传输。以下为不同类型的DDR命令：

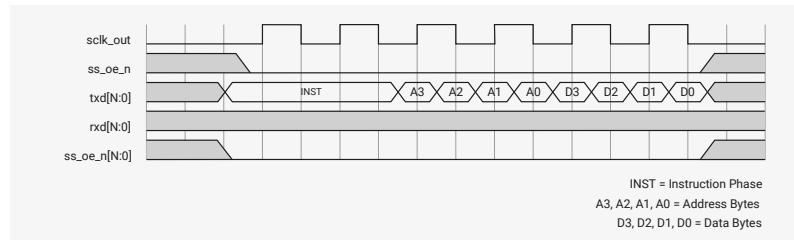
- 地址和数据以DDR格式传输（数据则为接收），而指令以标准格式传输。
- 指令、地址与数据均以DDR格式传输或接收。

DDR_EN (SPI_CTRLR0[16]) 位用于确定地址和数据是否以DDR模式传输，INST_DDR_EN (SPI_CTRLR0[17]) 位用于确定指令是否必须以DDR格式传输。上述位

are only valid when the CTRLR0.SPI_FRF bit is set to be in Dual, or Quad mode.

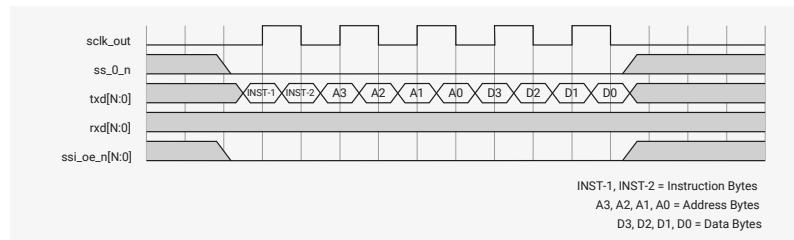
[Figure 158](#) describes a DDR write transfer where instructions are continued to be transmitted in standard format. In [Figure 158](#), the value of N will be 7 if CTRLR0.SPI_FRF is set to [11b](#), 3 if CTRLR0.SPI_FRF is set to [10b](#), and 1 if CTRLR0.SPI_FRF is set to [01b](#).

Figure 158. DDR Transfer with SCPH=0 and SCPOL=0



[Figure 159](#) describes a DDR write transfer where instruction, address and data all are transferred in DDR format.

Figure 159. DDR Transfer with Instruction, Address and Data Transmitted in DDR Format



NOTE

In the DDR transfer, address and instruction cannot be programmed to a value of 0.

4.10.10.5.1. Transmitting Data in DDR Mode

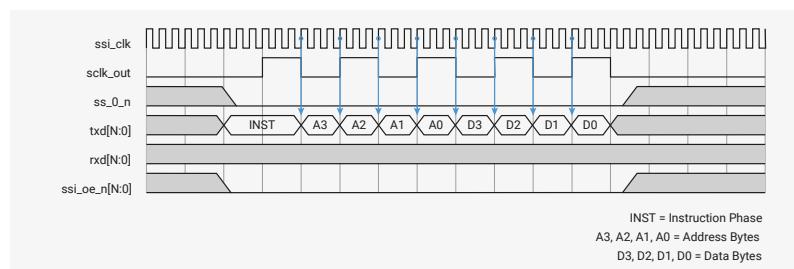
In DDR mode, data is transmitted on both edges so that it is difficult to sample data correctly. DW_apb_ssi uses an internal register to determine the edge on which the data should be transmitted. This will ensure that the receiver is able to get a stable data while sampling. The internal register (DDR_DRIVE_EDGE) determines the edge on which the data is transmitted. DW_apb_ssi sends data with respect to baud clock, which is an integral multiple of the internal clock (ssi_clk * BAUDR). The data needs to be transmitted within half clock cycle (BAUDR/2), therefore the maximum value for DDR_DRIVE_EDGE is equal to [(BAUDR/2)-1]. If the programmed value of DDR_DRIVE_EDGE is 0 then data is transmitted edge-aligned with respect to sclk_out (baud clock). If the programmed value of DDR_DRIVE_EDGE is one then the data is transmitted one ssi_clk before the edge of sclk_out.

NOTE

If the baud rate is programmed to be two, then the data will always be edge aligned.

[Figure 160](#), [Figure 161](#), and [Figure 162](#) show examples of how data is transmitted using different values of the DDR_DRIVE_EDGE register. The green arrows in these examples represent the points where data is driven. Baud rate used in all these examples is 12. In [Figure 160](#), transmit edge and driving edge of the data are the same. This is default behavior in DDR mode.

Figure 160. Transmit Data With DDR_DRIVE_EDGE = 0



仅当 CTRLR0.SPI_FRF 位设置为双模式或四模式时有效。

图158描述了一种DDR写传输，其中指令持续以标准格式传输。在图158中，当 CTRLR0.SPI_FRF 设置为 11b时，N 的值为 7；设置为 10b时，N 的值为3；设置为 01b时，N 的值为1。

图158。 DDR
传输， SCPOL=0
且 SCPOL=0

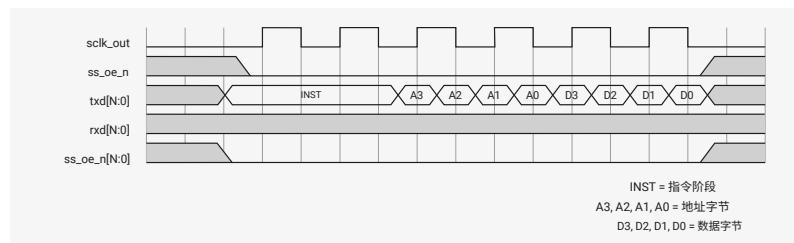
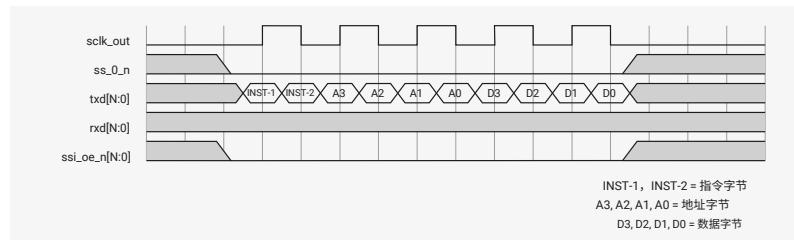


图159描述了一种DDR写传输，其中指令、地址和数据均以DDR格式传输。

图159。 指令
、地址及数
据以DDR格式传输



注意

在DDR传输中，地址和指令不能设置为0值。

4.10.10.5.1. DDR模式下的数据传输

在DDR模式下，数据在两个时钟边沿传输，因此难以准确采样数据。DW_apb_ssi使用内部寄存器确定数据应传输的时钟边沿。此设置确保接收端在采样时能获得稳定的数据。内部寄存器（DDR_DRIVE_EDGE）决定数据传输的时钟边沿。DW_apb_ssi根据波特率时钟发送数据，波特率时钟是内部时钟（ssi_clk * BAUDR）的整数倍。数据需在半个时钟周期（BAUDR/2）内传输，故DDR_DRIVE_EDGE的最大值为[(BAUDR/2)-1]。若 DDR_DRIVE_EDGE 编程值为 0，则数据相对于 sclk_out（波特率时钟）边沿进行传输对齐。若 DDR_DRIVE_EDGE 编程值为 1，则数据将在 sclk_out 边沿前一个 ssi_clk 时钟周期传输。

注意

若波特率被设定为 2，则数据始终边沿对齐。

图 160、图 161 及图 162 展示了使用不同 DDR_DRIVE_EDGE 寄存器值进行数据传输的示例。这些示例中的绿色箭头头标示数据驱动点。所有示例中使用的波特率均为 12。在图 160 中，数据传输边沿与驱动边沿一致。此为 DDR 模式下的默认行为。

图 160。 DDR_DRIVE
EDGE =
0时发送数据

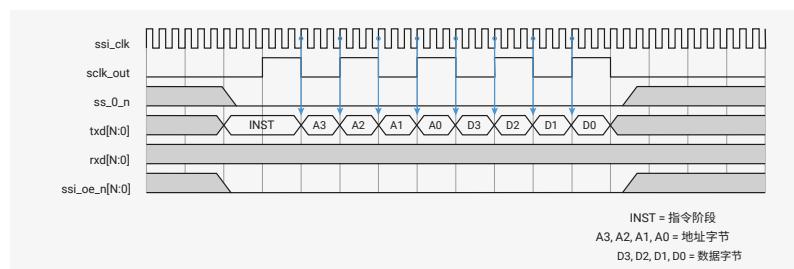


Figure 160 shows the default behavior in which the transmit and driving edge of the data is the same.

Figure 161. Transmit Data With
DDR_DRIVE_EDGE = 1

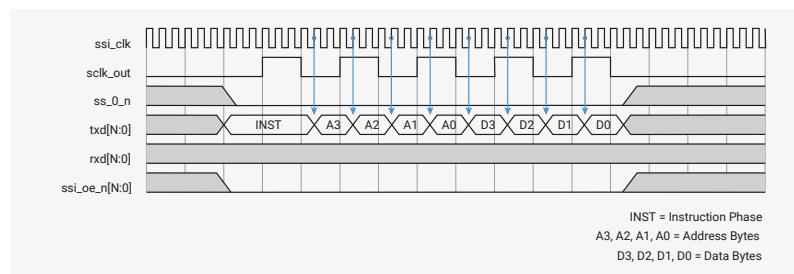
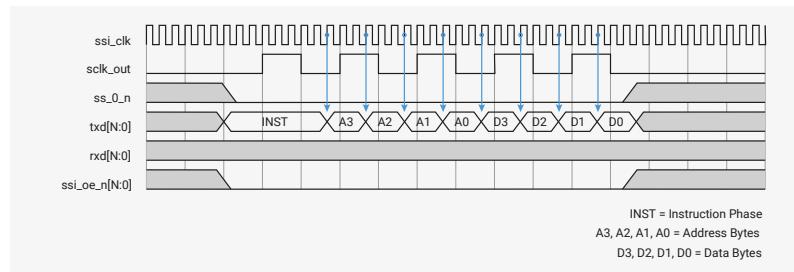


Figure 162. Transmit Data With
DDR_DRIVE_EDGE = 2



4.10.10.6. XIP Mode Support in SPI Mode

The eXecute In Place (XIP) mode enables transfer of SPI data directly through the APB interface without writing the data register of DW_apb_ssi. XIP mode is enabled in DW_apb_ssi when the XIP cache is enabled. This control signal indicates whether APB transfers are register read-write or XIP reads. When in XIP mode, DW_apb_ssi expects only read request on the APB interface. This request is translated to SPI read on the serial interface and soon after the data is received, the data is returned to the APB interface in the same transaction.

i NOTE

- Only APB reads are supported during an XIP operation

The address length is derived from the SPI_CTRLR0.ADDR_L field, and relevant bits from paddr ([SPI_CTRLR0.ADDR_L:1:0]) are transferred as address to the SPI interface. XIP address is managed by the XIP cache controller.

4.10.10.6.1. Read Operation in XIP Mode

The XIP operation is supported only in enhanced SPI modes (Dual, Quad) of operation. Therefore, the CTRLR0.SPI_FRF bit should not be programmed to 0. An XIP read operation is divided into two phases:

- Address phase
- Data phase

For an XIP read operation

- Set the SPI frame format and data frame size value in CTRLR0 register. Note that the value of the maximum data frame size is 32.
- Set the Address length, Wait cycles, and transaction type in the SPI_CTRLR0 register. Note that the maximum address length is 32.

After these settings, a user can initiate a read transaction through the APB interface which will be transferred to SPI peripheral using programmed values. Figure 163 shows the typical XIP transfer. The Value of N = 1, 3 and 7 for SPI mode Dual, and Quad modes, respectively.

图 160 显示了传输边沿与驱动边沿一致的默认行为。

图 161。DDR_DRIV

E_EDGE =
1时发送数据

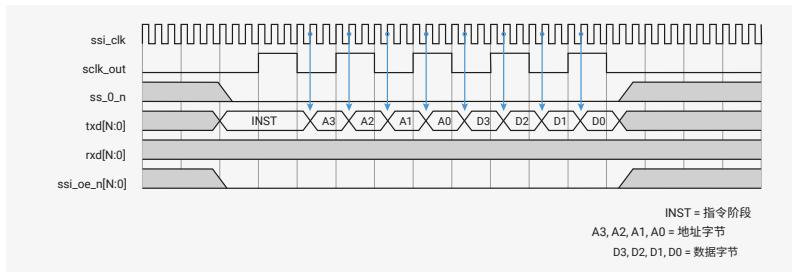
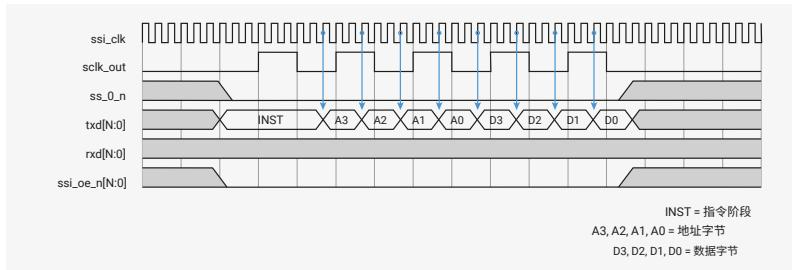


图 162。DDR_DRIV

E_EDGE =
2时发送数据



4.10.10.6. SPI 模式下的 XIP 模式支持

就地执行 (eXecute In Place, XIP) 模式允许通过 APB 接口直接传输 SPI 数据，而无需写入 DW_apb_ssi 的数据寄存器。当启用 XIP 缓存时，DW_apb_ssi 中将启用 XIP 模式。该控制信号指示 APB 传输是寄存器读写还是 XIP 读取操作。处于 XIP 模式时，DW_apb_ssi 仅在 APB 接口接收读请求。该请求将被转换为串行接口上的 SPI 读，数据接收后立即以同一事务返回 APB 接口。

① 注意

- XIP 操作期间仅支持 APB 读取。

地址长度由 SPI_CTRLR0.ADDR_L 字段确定，paddr 的相关位 ([SPI_CTRLR0.ADDR_L-1:0]) 作为地址传输至 SPI 接口。XIP 地址由 XIP 缓存控制器管理。

4.10.10.6.1 XIP 模式下的读操作

XIP 操作仅支持增强型 SPI 模式（双线和四线模式）下的操作。因此，CTRLR0.SPI_FRF 位不得被设置为 0。XIP 读操作分为两个阶段：

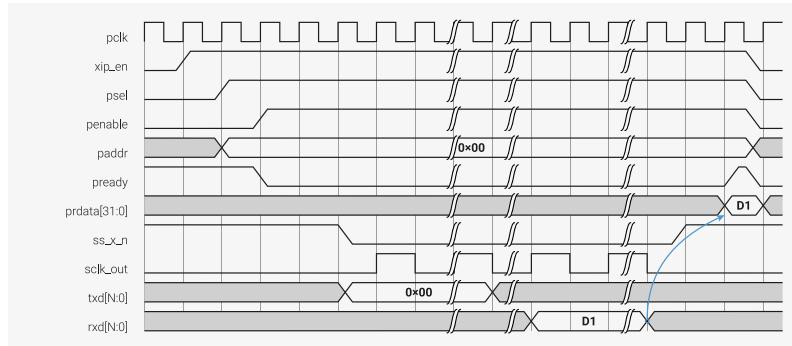
- 地址阶段
- 数据阶段

针对 XIP 读操作

1. 在 CTRLR0 寄存器中设置 SPI 帧格式和数据帧大小。注意，最大数据帧大小为 32。
2. 在 SPI_CTRLR0 寄存器中设置地址长度、等待周期及事务类型。注意，最大地址长度为 32。

完成上述设置后，用户可通过 APB 接口发起读事务，该事务将基于预设值传输至 SPI 外设。图 163 显示典型的 XIP 传输过程。N 的取值分别为：双线 SPI 模式为 1，四线 SPI 模式为 3 和 7。

Figure 163. Typical Read Operation in XIP Mode



4.10.11. DMA Controller Interface

The DW_apb_ssi has built-in DMA capability; it has a handshaking interface to a DMA Controller to request and control transfers. The APB bus is used to perform the data transfer to or from the DMA.

NOTE

When the DW_apb_ssi interfaces to the DMA controller, the DMA controller is always a flow controller; that is, it controls the block size. This must be programmed by software in the DMA controller.

The DW_apb_ssi uses two DMA channels, one for the transmit data and one for the receive data. The DW_apb_ssi has these DMA registers:

DMACR

Control register to enable DMA operation.

DMATDLR

Register to set the transmit FIFO level at which a DMA request is made.

DMARDLR

Register to set the receive FIFO level at which a DMA request is made.

The DW_apb_ssi uses the following handshaking signals to interface with the DMA controller.

- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req
- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req

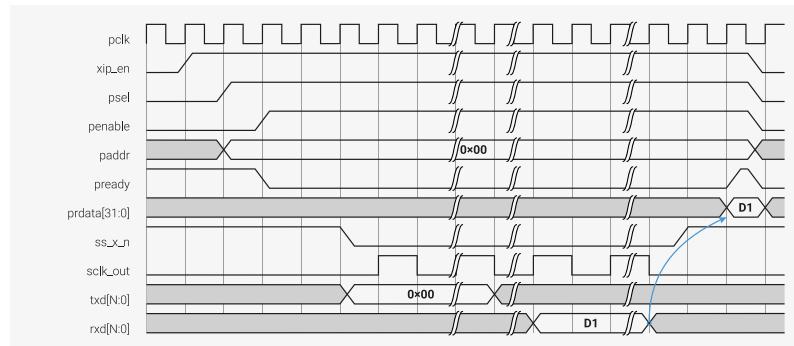
To enable the DMA Controller interface on the DW_apb_ssi, you must write the DMA Control Register (DMACR). Writing a 1 into the TDMAE bit field of DMACR register enables the DW_apb_ssi transmit handshaking interface. Writing a 1 into the RDMAE bit field of the DMACR register enables the DW_apb_ssi receive handshaking interface.

Table 580 provides description for different DMA transmit data level values.

Table 580. DMA Transmit Data Level (DMATDL) Decode Value

DMATDL Value	Description
0000_0000	dma_tx_req is asserted when zero data entries are present in the transmit FIFO
0000_0001	dma_tx_req is asserted when one or less data entry is present in the transmit FIFO

图163。典型示例
XIP中的读取操作
模式



4.10.11. DMA 控制器接口

DW_apb_ssi具备内置的DMA功能；其具备与DMA控制器通信的握手接口，用以请求和控制数据传输。APB总线用于执行与DMA之间的数据传输。

i 注意

当DW_apb_ssi与DMA控制器连接时，DMA控制器始终作为流量控制器；即负责控制块的大小。该功能必须由DMA控制器中的软件编程实现。

DW_apb_ssi使用两个DMA通道，分别用于发送和接收数据。DW_apb_ssi包含以下DMA寄存器：

DMACR

用于启用DMA操作的控制寄存器。

DMATDLR

用于设置发送FIFO达到该级别时发出DMA请求的寄存器。

DMARDLR

用于设置接收FIFO达到该级别时发出DMA请求的寄存器。

DW_apb_ssi通过以下握手信号与DMA控制器接口。

- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req
- dma_tx_req
- dma_tx_single
- dma_tx_ack
- dma_rx_req

要使能 DW_apb_ssi 上的 DMA 控制器接口，必须写入 DMA 控制寄存器（DMACR）。向 DMACR 寄存器的 TDMAE 位域写入1可使能 DW_apb_ssi 传输握手接口。向 DMACR 寄存器的 RDMAE 位域写入1可使能 DW_apb_ssi 接收握手接口。

[表580 提供了不同 DMA 传输数据级别值的说明。](#)

表580。DMA
传输数据级别
(DMATDL) 解码
值

DMATDL 值	描述
0000_0000	当传输 FIFO 中无数据项时，dma_tx_req 置位。
0000_0001	当传输 FIFO 中数据项少于或等于1时，dma_tx_req 置位。

0000_0010	dma_tx_req is asserted when two or less data entries are present in the transmit FIFO
...	...
0000_1101	dma_tx_req is asserted when 13 or less data entries are present in the transmit FIFO
0000_1110	dma_tx_req is asserted when 14 or less data entries are present in the transmit FIFO
0000_1111	dma_tx_req is asserted when 15 or less data entries are present in the transmit FIFO

Table 581 provides description for different DMA Receive Data Level values.

Table 581. DMA Receive Data Level (DMARDL) Decode Value

DMARDL Value	Description
0000_0000	dma_rx_req is asserted when one or more data entries are present in the receive FIFO
0000_0001	dma_rx_req is asserted when two or more data entries are present in the receive FIFO
0000_0010	dma_rx_req is asserted when three or more data entries are present in the receive FIFO
...	...
0000_1101	dma_rx_req is asserted when 14 or more data entries are present in the receive FIFO
0000_1110	dma_rx_req is asserted when 15 or more data entries are present in the receive FIFO
0000_1111	dma_rx_req is asserted when 16 data entries are present in the receive FIFO

4.10.11.1. Overview of Operation

As a block flow control device, the DMA Controller is programmed by the processor with the number of data items (block size) that are to be transmitted or received by the DW_apb_ssi.

The block is broken into a number of transactions, each initiated by a request from the DW_apb_ssi. The DMA Controller must also be programmed with the number of data items (in this case, DW_apb_ssi FIFO entries) to be transferred for each DMA request. This is also known as the burst transaction length.

Figure 164 shows a single block transfer, where the block size programmed into the DMA Controller is 12 and the burst transaction length is set to four. In this case, the block size is a multiple of the burst transaction length; therefore, the DMA block transfer consists of a series of burst transactions.

⚠ CAUTION

On RP2040, the burst transaction length of the SSI's DMA interface is fixed at four transfers. **SSI.DMARDLR** must always be equal to 4, which is the value it takes at reset. The SSI will then request a single transfer when it has between one and three items in its FIFO, and a 4-burst when it has four or more.

0000_0010	当传输 FIFO 中数据项少于或等于2时, dma_tx_req 置位。
...	...
0000_1101	当传输 FIFO 中数据项少于或等于13时, dma_tx_req 置位。
0000_1110	当传输 FIFO 中数据项少于或等于14时, dma_tx_req 置位。
0000_1111	当发送 FIFO 中存在 15 个或更少数据条目时, dma_tx_req 被置位

表 581 对不同 DMA 接收数据级别值进行了说明。

表 581。DMA
接收数据级别
(DMARDL) 解码
值

DMARDL 值	描述
0000_0000	当接收 FIFO 中存在一个或多个数据条目时, dma_rx_req 被置位
0000_0001	当接收 FIFO 中存在两个或多个数据条目时, dma_rx_req 被置位
0000_0010	当接收 FIFO 中存在三个或多个数据条目时, dma_rx_req 被置位
...	...
0000_1101	当接收 FIFO 中存在 14 个或多个数据条目时, dma_rx_req 被置位
0000_1110	当接收 FIFO 中存在 15 个或多个数据条目时, dma_rx_req 被置位
0000_1111	当接收 FIFO 中存在 16 个数据条目时, dma_rx_req 被置位

4.10.11.1. 运行概述

作为块流控设备, DMA 控制器由处理器编程设定通过 DW_apb_ssi 传输或接收的数据项数量 (块大小)。

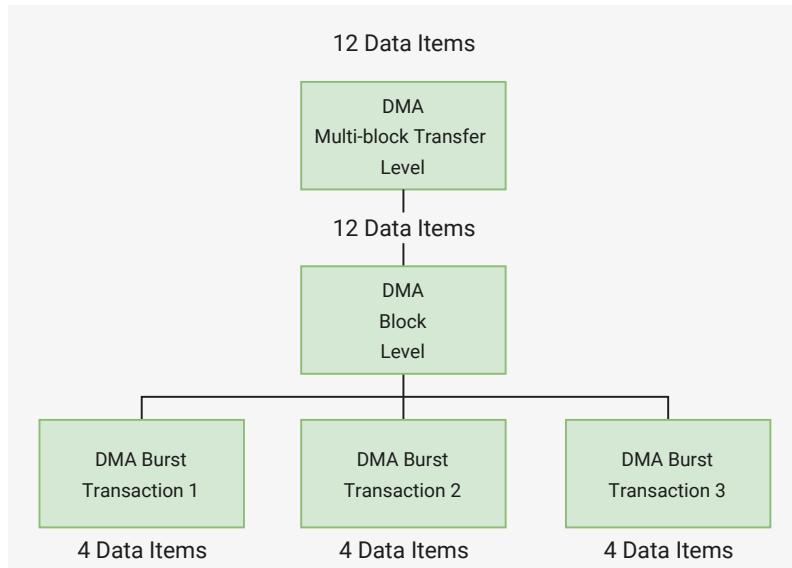
该数据块被划分为多个事务, 每个事务由DW_apb_ssi发起请求。DMA控制器还必须配置每个DMA请求所传输的数据项数量 (本例中为DW_apb_ssi FIFO条目数)。此长度亦称为突发事务长度。

图164展示了单个块传输, DMA控制器中配置的块大小为12, 突发事务长度设置为4。在此情况下, 块大小为突发事务长度的整数倍; 因此, DMA块传输由一系列突发事务组成。

⚠ 注意

在RP2040上, SSI的DMA接口突发事务长度固定为4次传输。SSI.DMARDLR必须始终设为4, 此为复位时的默认值。当SSI的FIFO内有1至3个条目时, SSI将请求单次传输; 当有4个或以上条目时, 则请求4次突发传输。

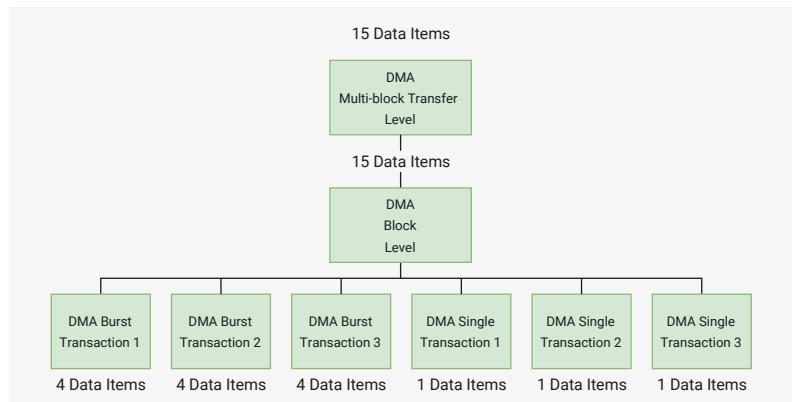
Figure 164.
Breakdown of DMA Transfer into Burst Transactions. Block size,
`DMA.CTLx.BLOCKS_TS = 12. Number of data items per source burst transaction,`
`DMA.CTLx.SRC_MSIZE = 4. SSI receive FIFO watermark level,`
`SSI.DMARDLR + 1 = DMA.CTLx.SRC_MSIZE = 4`



If the DW_apb_ssi makes a transmit request to this channel, four data items are written to the DW_apb_ssi transmit FIFO. Similarly, if the DW_apb_ssi makes a receive request to this channel, four data items are read from the DW_apb_ssi receive FIFO. Three separate requests must be made to this DMA channel before all 12 data items are written or read.

When the block size programmed into the DMA Controller is not a multiple of the burst transaction length, as shown in [Figure 165](#), a series of burst transactions followed by single transactions are needed to complete the block transfer.

Figure 165.
Breakdown of DMA Transfer into Single and Burst Transactions. Block size,
`DMA.CTLx.BLOCK_TS = 15. Number of data items per burst transaction,`
`DMA.CTLx.DEST_MSIZE = 4. SSI transmit FIFO watermark level,`
`SSI.DMATDLR = DMA.CTLx.DEST_MSIZE = 4`



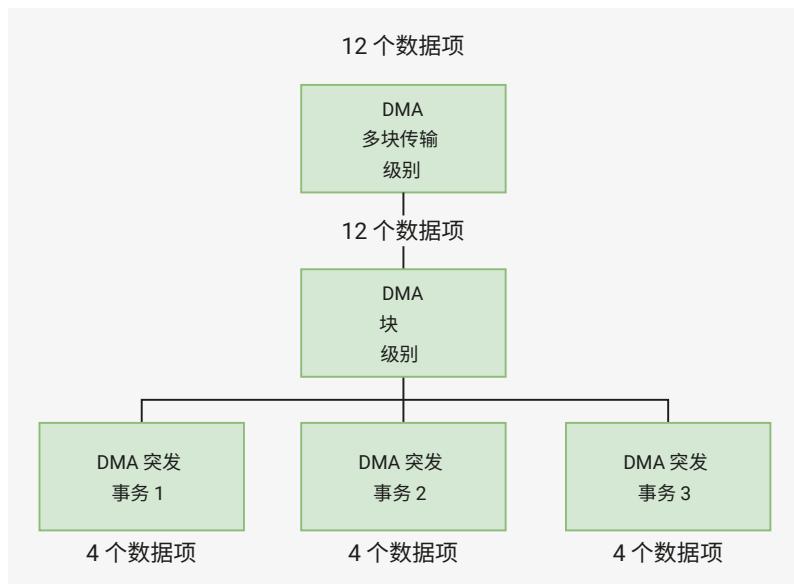
4.10.12. APB Interface

The host processor accesses data, control, and status information on the DW_apb_ssi through the APB interface. APB accesses to the DW_apb_ssi peripheral are described in the following subsections.

4.10.12.1. Control and Status Register APB Access

Control and status registers within the DW_apb_ssi are byte-addressable. The maximum width of the control or status register in the DW_apb_ssi is 16 bits. Therefore all read and write operations to the DW_apb_ssi control and status registers require only one APB access.

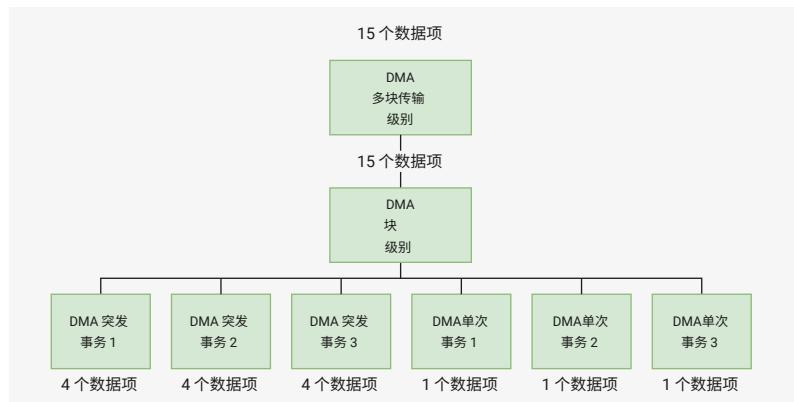
图 164。
DMA 传输拆分为
突发事务。块大
小, DMA.CTLx.BL0
CKS_
TS = 12。每个源突
发事务中的数据
项数量, DMA.CTLx.SR
C_MSIZEx4
SSI 接收 FIFO 水
位线级别, SSI.D
MARDLR+1 = DMA
.CTLx.SRC_MSIZEx4



若 DW_apb_ssi 向该通道发出传输请求，则写入四个数据项至 DW_apb_ssi 传输 FIFO。同理，若 DW_apb_ssi 向该通道发出接收请求，则从 DW_apb_ssi 接收 FIFO 读取四个数据项。须对该 DMA 通道发起三次独立请求，方可完成全部 12 个数据项的写入或读取。

当编程至 DMA 控制器的块大小不是突发事务长度的整数倍时，如图 165 所示，需要通过一系列突发事务及后续单次事务完成块传输。

图 165。
DMA 传输分解为
单次事务和突发
事务。
块大小, DMA.CTLx.BL
OCK_TS = 15。每个
突发事务中的数
据项数量,
DMA.CTLx.DEST_MSIZEx4
= 4. SSI 发送 FIFO
水位线级别,
SSI.DMADLR =
DMA.CTLx.DEST_MSIZEx4
= 4



4.10.12. APB 接口

主处理器通过 APB 接口访问 DW_apb_ssi 的数据、控制及状态信息。对 DW_apb_ssi 外设的 APB 访问在以下小节中详细描述。

4.10.12.1. 控制和状态寄存器的 APB 访问

DW_apb_ssi 中的控制与状态寄存器支持字节寻址。DW_apb_ssi 中控制或状态寄存器的最大位宽为 16 位。因此，对 DW_apb_ssi 控制和状态寄存器的所有读写操作仅需一次 APB 访问。

4.10.12.2. Data Register APB Access

The data register (DR) within the DW_apb_ssi is 32 bits wide in order to remain consistent with the maximum serial transfer size (data frame). An APB write operation to DR moves data from pwdata into the transmit FIFO buffer. An APB read operation from DR moves data from the receive FIFO buffer onto prdata.

The DW_apb_ssi DR can be written/read in one APB access.

NOTE

The DR register in the DW_apb_ssi occupies sixty-four 32-bit locations of the memory map to facilitate AHB burst transfers. There are no burst transactions on the APB bus itself, but DW_apb_ssi supports the AHB bursts that happen on the AHB side of the AHB/APB bridge. Writing to any of these address locations has the same effect as pushing the data from the pwdata bus into the transmit FIFO. Reading from any of these locations has the same effect as popping data from the receive FIFO onto the prdata bus. The FIFO buffers on the DW_apb_ssi are not addressable.

4.10.13. List of Registers

The SSI registers start at a base address of `0x18000000` (defined as `XIP_SSI_BASE` in SDK).

Table 582. List of SSI registers

Offset	Name	Info
0x00	CTRLR0	Control register 0
0x04	CTRLR1	Master Control register 1
0x08	SSIENR	SSI Enable
0x0c	MWCR	Microwire Control
0x10	SER	Slave enable
0x14	BAUDR	Baud rate
0x18	TXFTLR	TX FIFO threshold level
0x1c	RXFTLR	RX FIFO threshold level
0x20	TXFLR	TX FIFO level
0x24	RXFLR	RX FIFO level
0x28	SR	Status register
0x2c	IMR	Interrupt mask
0x30	ISR	Interrupt status
0x34	RISR	Raw interrupt status
0x38	TXOICR	TX FIFO overflow interrupt clear
0x3c	RXOICR	RX FIFO overflow interrupt clear
0x40	RXUICR	RX FIFO underflow interrupt clear
0x44	MSTICR	Multi-master interrupt clear
0x48	ICR	Interrupt clear
0x4c	DMACR	DMA control
0x50	DMATDLR	DMA TX data level
0x54	DMARDLR	DMA RX data level

4.10.12.2. 数据寄存器APB访问

DW_apb_ssi中的数据寄存器（DR）宽度为32位，以保持与最大串行传输大小（数据帧）的一致性。对DR进行APB写操作时，数据将从pdata传输至发送FIFO缓冲区。对DR进行APB读操作时，数据将从接收FIFO缓冲区传输至pdata。

DW_apb_ssi的DR寄存器可通过一次APB访问完成写入和读取。

注意

DW_apb_ssi中的DR寄存器在内存映射中占用64个32位位置，以支持AHB突发传输。APB总线本身不支持突发传输，但DW_apb_ssi支持发生在AHB/APB桥接器AHB侧的AHB突发传输。写入上述任一地址位置的操作，效果等同于将数据从pdata总线推入发送FIFO；从上述任一地址读取数据的操作，效果等同于将数据从接收FIFO弹出至pdata总线。DW_apb_ssi上的FIFO缓冲区不可寻址。

4.10.13. 寄存器列表

SSI寄存器起始基址为 `0x18000000`（在SDK中定义为XIP_SSI_BASE）。

表582 SSI寄存器列表

偏移量	名称	说明
0x00	CTRLR0	控制寄存器0
0x04	CTRLR1	主控寄存器1
0x08	SSIENR	SSI使能
0x0c	MWCR	Microwire控制
0x10	SER	从设备使能
0x14	BAUDR	波特率
0x18	TXFTLR	TX FIFO阈值电平
0x1c	RXFTLR	RX FIFO阈值电平
0x20	TXFLR	发射FIFO级别
0x24	RXFLR	接收FIFO级别
0x28	SR	状态寄存器
0x2c	IMR	中断屏蔽
0x30	ISR	中断状态
0x34	RISR	原始中断状态
0x38	TXOICR	发射FIFO溢出中断清除
0x3c	RXOICR	接收FIFO溢出中断清除
0x40	RXUICR	接收FIFO欠流中断清除
0x44	MSTICR	多主中断清除
0x48	ICR	中断清除
0x4c	DMACR	DMA 控制
0x50	DMATDLR	DMA TX 数据级别
0x54	DMARDLR	DMA RX 数据级别

Offset	Name	Info
0x58	IDR	Identification register
0x5c	SSI_VERSION_ID	Version ID
0x60	DRO	Data Register 0 (of 36)
0xf0	RX_SAMPLE_DLY	RX sample delay
0xf4	SPI_CTRLR0	SPI control
0xf8	TXD_DRIVE_EDGE	TX drive edge

SSI: CTRLR0 Register

Offset: 0x00

Description

Control register 0

Table 583. CTRLR0 Register

Bits	Description	Type	Reset
31:25	Reserved.	-	-
24	SSTE : Slave select toggle enable	RW	0x0
23	Reserved.	-	-
22:21	SPI_FRF : SPI frame format	RW	0x0
	Enumerated values:		
	0x0 → STD: Standard 1-bit SPI frame format; 1 bit per SCK, full-duplex		
	0x1 → DUAL: Dual-SPI frame format; two bits per SCK, half-duplex		
	0x2 → QUAD: Quad-SPI frame format; four bits per SCK, half-duplex		
20:16	DFS_32 : Data frame size in 32b transfer mode Value of n → n+1 clocks per frame.	RW	0x00
15:12	CFS : Control frame size Value of n → n+1 clocks per frame.	RW	0x0
11	SRL : Shift register loop (test mode)	RW	0x0
10	SLV_OE : Slave output enable	RW	0x0
9:8	TMOD : Transfer mode	RW	0x0
	Enumerated values:		
	0x0 → TX_AND_RX: Both transmit and receive		
	0x1 → TX_ONLY: Transmit only (not for FRF == 0, standard SPI mode)		
	0x2 → RX_ONLY: Receive only (not for FRF == 0, standard SPI mode)		
	0x3 → EEPROM_READ: EEPROM read mode (TX then RX; RX starts after control data TX'd)		
7	SCPOL : Serial clock polarity	RW	0x0
6	SCPH : Serial clock phase	RW	0x0
5:4	FRF : Frame format	RW	0x0
3:0	DFS : Data frame size	RW	0x0

偏移量	名称	说明
0x58	IDR	标识寄存器
0x5c	SSI_VERSION_ID	版本 ID
0x60	DRO	数据寄存器 0 (共36个)
0xf0	RX_SAMPLE_DLY	接收采样延迟
0xf4	SPI_CTRLR0	SPI 控制
0xf8	TXD_DRIVE_EDGE	TX 驱动边缘

SSI：CTRLR0 寄存器

偏移: 0x00

描述

控制寄存器0

表 583。CTRLR0 寄存器

位	描述	类型	复位值
31:25	保留。	-	-
24	SSTE : 从选择切换使能	读写	0x0
23	保留。	-	-
22:21	SPI_FRF : SPI 帧格式	读写	0x0
	枚举值:		
	0x0 → STD: 标准 1 位 SPI 帧格式; 每个 SCK 1 位, 全双工		
	0x1 → DUAL: 双重 SPI 帧格式; 每个 SCK 两位, 半双工		
	0x2 → QUAD: 四重 SPI 帧格式; 每个 SCK 四位, 半双工		
20:16	DFS_32 : 32 位传输模式下的数据帧大小 n 值 → 每帧 n+1 个时钟	读写	0x00
15:12	CFS : 控制帧大小 n 值 → 每帧 n+1 个时钟	读写	0x0
11	SRL : 移位寄存器环 (测试模式)	读写	0x0
10	SLV_OE : 从机输出使能	读写	0x0
9:8	TMOD : 传输模式	读写	0x0
	枚举值:		
	0x0 → TX_AND_RX: 同时发射和接收		
	0x1 → TX_ONLY: 仅发射 (不适用于 FRF == 0, 即标准 SPI 模式)		
	0x2 → RX_ONLY: 仅接收 (不适用于 FRF == 0, 即标准 SPI 模式)		
	0x3 → EEPROM_READ: EEPROM 读取模式 (先发送 TX, 后接收 RX; RX 于控制数据 TX 完成后开始)		
7	SCPOL : 串行时钟极性	读写	0x0
6	SCPH : 串行时钟相位	读写	0x0
5:4	FRF : 帧格式	读写	0x0
3:0	DFS : 数据帧大小	读写	0x0

SSI: CTRLR1 Register

Offset: 0x04

Description

Master Control register 1

Table 584. CTRLR1 Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	NDF: Number of data frames	RW	0x0000

SSI: SSIENR Register

Offset: 0x08

Description

SSI Enable

Table 585. SSIENR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	SSI_EN: SSI enable	RW	0x0

SSI: MWCR Register

Offset: 0x0c

Description

Microwire Control

Table 586. MWCR Register

Bits	Description	Type	Reset
31:3	Reserved.	-	-
2	MHS: Microwire handshaking	RW	0x0
1	MDD: Microwire control	RW	0x0
0	MWMOD: Microwire transfer mode	RW	0x0

SSI: SER Register

Offset: 0x10

Description

Slave enable

Table 587. SER Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	For each bit: 0 → slave not selected 1 → slave selected	RW	0x0

SSI: BAUDR Register

Offset: 0x14

Description

Baud rate

SSI: CTRLR1寄存器

偏移: 0x04

描述

主控寄存器1

表584. CTRLR1寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	NDF : 数据帧数量	读写	0x0000

SSI: SSIENR寄存器

偏移: 0x08

描述

SSI使能

表585. SSIENR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	SSI_EN : SSI使能	读写	0x0

SSI: MWCR寄存器

偏移: 0x0c

描述

Microwire控制

表586. MWCR寄存器

位	描述	类型	复位值
31:3	保留。	-	-
2	MHS : Microwire握手	读写	0x0
1	MDD : Microwire控制	读写	0x0
0	MWMOD : Microwire传输模式	读写	0x0

SSI: SER寄存器

偏移: 0x10

描述

从属使能

表587. SER寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	针对每个位: 0 → 未选中从属 1 → 选中从属	读写	0x0

SSI: BAUDR 寄存器

偏移: 0x14

描述

波特率

Table 588. BAUDR Register

Bits	Description	Type	Reset
31:16	Reserved.	-	-
15:0	SCKDV : SSI clock divider	RW	0x0000

SSI: TXFTLR Register

Offset: 0x18

Description

TX FIFO threshold level

Table 589. TXFTLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	TFT : Transmit FIFO threshold	RW	0x00

SSI: RXFTLR Register

Offset: 0x1c

Description

RX FIFO threshold level

Table 590. RXFTLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	RFT : Receive FIFO threshold	RW	0x00

SSI: TXFLR Register

Offset: 0x20

Description

TX FIFO level

Table 591. TXFLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	TFTFL : Transmit FIFO level	RO	0x00

SSI: RXFLR Register

Offset: 0x24

Description

RX FIFO level

Table 592. RXFLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	RXTFL : Receive FIFO level	RO	0x00

SSI: SR Register

Offset: 0x28

表 588. BAUDR
寄存器

位	描述	类型	复位值
31:16	保留。	-	-
15:0	SCKDV : SSI 时钟分频器	读写	0x0000

SSI: TXFTLR 寄存器

偏移: 0x18

描述

TX FIFO 阈值电平

表 589. TXFTLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TFT : 发送 FIFO 阈值	读写	0x00

SSI: RXFTLR 寄存器

偏移量: 0x1c

描述

RX FIFO 阈值电平

表 590. RXFTLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RFT : 接收 FIFO 阈值	读写	0x00

SSI: TXFLR 寄存器

偏移: 0x20

描述

发送 FIFO 水平

表 591. TXFLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	TFTFL : 发送FIFO级别	只读	0x00

SSI: RXFLR 寄存器

偏移: 0x24

描述

接收FIFO级别

表 592. RXFLR
寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RXTFL : 接收FIFO级别	只读	0x00

SSI: SR 寄存器

偏移: 0x28

Description

Status register

Table 593. SR Register

Bits	Description	Type	Reset
31:7	Reserved.	-	-
6	DCOL: Data collision error	RO	0x0
5	TXE: Transmission error	RO	0x0
4	RFF: Receive FIFO full	RO	0x0
3	RFNE: Receive FIFO not empty	RO	0x0
2	TFE: Transmit FIFO empty	RO	0x0
1	TFNF: Transmit FIFO not full	RO	0x0
0	BUSY: SSI busy flag	RO	0x0

SSI: IMR Register

Offset: 0x2c

Description

Interrupt mask

Table 594. IMR Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5	MSTIM: Multi-master contention interrupt mask	RW	0x0
4	RXFIM: Receive FIFO full interrupt mask	RW	0x0
3	RXOIM: Receive FIFO overflow interrupt mask	RW	0x0
2	RXUIM: Receive FIFO underflow interrupt mask	RW	0x0
1	TXOIM: Transmit FIFO overflow interrupt mask	RW	0x0
0	TXEIM: Transmit FIFO empty interrupt mask	RW	0x0

SSI: ISR Register

Offset: 0x30

Description

Interrupt status

Table 595. ISR Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5	MSTIS: Multi-master contention interrupt status	RO	0x0
4	RXFIS: Receive FIFO full interrupt status	RO	0x0
3	RXOIS: Receive FIFO overflow interrupt status	RO	0x0
2	RXUIS: Receive FIFO underflow interrupt status	RO	0x0
1	TXOIS: Transmit FIFO overflow interrupt status	RO	0x0
0	TXEIS: Transmit FIFO empty interrupt status	RO	0x0

描述

状态寄存器

表593。SR寄存器

位	描述	类型	复位值
31:7	保留。	-	-
6	DCOL : 数据冲突错误	只读	0x0
5	TXE : 传输错误	只读	0x0
4	RFF : 接收FIFO满	只读	0x0
3	RFNE : 接收FIFO非空	只读	0x0
2	TFE : 发送FIFO空	只读	0x0
1	TFNF : 发送FIFO未满	只读	0x0
0	BUSY : SSI忙标志	只读	0x0

SSI：IMR寄存器

偏移: 0x2c

描述

中断屏蔽

表594。IMR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIM : 多主控争用中断屏蔽	读写	0x0
4	RXFIM : 接收FIFO满中断屏蔽	读写	0x0
3	RXOIM : 接收FIFO溢出中断屏蔽	读写	0x0
2	RXUIM : 接收FIFO欠载中断屏蔽	读写	0x0
1	TXOIM : 发送FIFO溢出中断屏蔽	读写	0x0
0	TXEIM : 发送FIFO空中断屏蔽	读写	0x0

SSI：ISR寄存器

偏移: 0x30

描述

中断状态

表595。ISR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIS : 多主控争用中断状态	只读	0x0
4	RXFIS : 接收FIFO满中断状态	只读	0x0
3	RXOIS : 接收FIFO溢出中断状态	只读	0x0
2	RXUIS : 接收FIFO欠载中断状态	只读	0x0
1	TXOIS : 发送FIFO溢出中断状态	只读	0x0
0	TXEIS : 发送FIFO空中断状态	只读	0x0

SSI: RISR Register

Offset: 0x34

Description

Raw interrupt status

Table 596. RISR Register

Bits	Description	Type	Reset
31:6	Reserved.	-	-
5	MSTIR: Multi-master contention raw interrupt status	RO	0x0
4	RXFIR: Receive FIFO full raw interrupt status	RO	0x0
3	RXOIR: Receive FIFO overflow raw interrupt status	RO	0x0
2	RXUIR: Receive FIFO underflow raw interrupt status	RO	0x0
1	TXOIR: Transmit FIFO overflow raw interrupt status	RO	0x0
0	TXEIR: Transmit FIFO empty raw interrupt status	RO	0x0

SSI: TXOICR Register

Offset: 0x38

Description

TX FIFO overflow interrupt clear

Table 597. TXOICR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Clear-on-read transmit FIFO overflow interrupt	RO	0x0

SSI: RXOICR Register

Offset: 0x3c

Description

RX FIFO overflow interrupt clear

Table 598. RXOICR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Clear-on-read receive FIFO overflow interrupt	RO	0x0

SSI: RXUICR Register

Offset: 0x40

Description

RX FIFO underflow interrupt clear

SSI：RISR寄存器

偏移: 0x34

描述

原始中断状态

表596. RISR寄存器

位	描述	类型	复位值
31:6	保留。	-	-
5	MSTIR : 多主控争用原始中断状态	只读	0x0
4	RXFIR : 接收FIFO已满原始中断状态	只读	0x0
3	RXOIR : 接收FIFO溢出原始中断状态	只读	0x0
2	RXUIR : 接收FIFO欠载原始中断状态	只读	0x0
1	TXOIR : 发送FIFO溢出原始中断状态	只读	0x0
0	TXEIR : 发送FIFO为空原始中断状态	只读	0x0

SSI：TXOICR寄存器

偏移: 0x38

描述

发射FIFO溢出中断清除

表597. TXOICR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除发送FIFO溢出中断	只读	0x0

SSI：RXOICR寄存器

偏移: 0x3c

描述

接收FIFO溢出中断清除

表598. RXOICR寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除接收FIFO溢出中断	只读	0x0

SSI：RXUICR寄存器

偏移: 0x40

描述

接收FIFO欠流中断清除

Table 599. RXUICR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Clear-on-read receive FIFO underflow interrupt	RO	0x0

SSI: MSTICR Register

Offset: 0x44

Description

Multi-master interrupt clear

Table 600. MSTICR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Clear-on-read multi-master contention interrupt	RO	0x0

SSI: ICR Register

Offset: 0x48

Description

Interrupt clear

Table 601. ICR Register

Bits	Description	Type	Reset
31:1	Reserved.	-	-
0	Clear-on-read all active interrupts	RO	0x0

SSI: DMACR Register

Offset: 0x4c

Description

DMA control

Table 602. DMACR Register

Bits	Description	Type	Reset
31:2	Reserved.	-	-
1	TDMAE: Transmit DMA enable	RW	0x0
0	RDMAE: Receive DMA enable	RW	0x0

SSI: DMATDLR Register

Offset: 0x50

Description

DMA TX data level

Table 603. DMATDLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	DMATDL: Transmit data watermark level	RW	0x00

SSI: DMARDLR Register

Offset: 0x54

表599. RXUICR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除接收FIFO欠载中断	只读	0x0

SSI：MSTICR寄存器

偏移: 0x44

描述

多主中断清除

表600. MSTICR
寄存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读清除多主控争用中断	只读	0x0

SSI：ICR 寄存器

偏移: 0x48

描述

中断清除

表601. ICR寄
存器

位	描述	类型	复位值
31:1	保留。	-	-
0	读取时清除所有激活的中断	只读	0x0

SSI：DMACR 寄存器

偏移: 0x4c

描述

DMA 控制

表602. DMACR寄
存器

位	描述	类型	复位值
31:2	保留。	-	-
1	TDMAE ：传输 DMA 使能	读写	0x0
0	RDMAE ：接收 DMA 使能	读写	0x0

SSI：DMATDLR 寄存器

偏移: 0x50

说明

DMA TX 数据级别

表603. DMATDLR寄
存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	DMATDL ：传输数据水位线	读写	0x00

SSI：DMARDLR 寄存器

偏移: 0x54

Description

DMA RX data level

Table 604. DMARDLR Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	DMARDL : Receive data watermark level (DMARDLR+1)	RW	0x00

SSI: IDR Register

Offset: 0x58

Description

Identification register

Table 605. IDR Register

Bits	Description	Type	Reset
31:0	IDCODE : Peripheral identification code	RO	0x51535049

SSI: SSI_VERSION_ID Register

Offset: 0x5c

Description

Version ID

Table 606. SSI_VERSION_ID Register

Bits	Description	Type	Reset
31:0	SSI_COMP_VERSION : SNPS component version (format X.YY)	RO	0x3430312a

SSI: DR0 Register

Offset: 0x60

Description

Data Register 0 (of 36)

Table 607. DR0 Register

Bits	Description	Type	Reset
31:0	DR : First data register of 36	RW	0x00000000

SSI: RX_SAMPLE_DLY Register

Offset: 0xf0

Description

RX sample delay

Table 608. RX_SAMPLE_DLY Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	RSD : RXD sample delay (in SCLK cycles)	RW	0x00

SSI: SPI_CTRLR0 Register

Offset: 0xf4

Description

SPI control

描述

DMA 接收数据级别

表604. DMARDLR寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	DMARDL : 接收数据水位线 (DMARDLR+1)	读写	0x00

SSI: IDR 寄存器

偏移: 0x58

描述

识别寄存器

表 605. IDR 寄存器

位	描述	类型	复位值
31:0	IDCODE : 外设标识码	只读	0x51535049

SSI: SSI_VERSION_ID 寄存器

偏移: 0x5c

描述

版本标识

表 606. SSI_VERSION_ID 寄存器

位	描述	类型	复位值
31:0	SSI_COMP_VERSION : SNPS 组件版本 (格式 X.YY)	只读	0x3430312a

SSI: DRO 寄存器

偏移: 0x60

描述

数据寄存器 0 (共36个)

表 607. DRO 寄存器

位	描述	类型	复位值
31:0	DR : 36 个数据寄存器中的第一个	读写	0x00000000

SSI: RX_SAMPLE_DLY 寄存器

偏移: 0xf0

描述

RX 采样延迟

表 608. RX_SAMPLE_DLY 寄存器

位	描述	类型	复位值
31:8	保留。	-	-
7:0	RSD : RXD 采样延迟 (SCLK 周期)	读写	0x00

SSI: SPI_CTRLR0 寄存器

偏移: 0xf4

描述

SPI控制

Table 609.
SPI_CTRLR0 Register

Bits	Description	Type	Reset
31:24	XIP_CMD : SPI Command to send in XIP mode (INST_L = 8-bit) or to append to Address (INST_L = 0-bit)	RW	0x03
23:19	Reserved.	-	-
18	SPI_RXDS_EN : Read data strobe enable	RW	0x0
17	INST_DDR_EN : Instruction DDR transfer enable	RW	0x0
16	SPI_DDR_EN : SPI DDR transfer enable	RW	0x0
15:11	WAIT_CYCLES : Wait cycles between control frame transmit and data reception (in SCLK cycles)	RW	0x00
10	Reserved.	-	-
9:8	INST_L : Instruction length (0/4/8/16b)	RW	0x0
	Enumerated values:		
	0x0 → NONE: No instruction		
	0x1 → 4B: 4-bit instruction		
	0x2 → 8B: 8-bit instruction		
	0x3 → 16B: 16-bit instruction		
7:6	Reserved.	-	-
5:2	ADDR_L : Address length (0b-60b in 4b increments)	RW	0x0
1:0	TRANS_TYPE : Address and instruction transfer format	RW	0x0
	Enumerated values:		
	0x0 → 1C1A: Command and address both in standard SPI frame format		
	0x1 → 1C2A: Command in standard SPI format, address in format specified by FRF		
	0x2 → 2C2A: Command and address both in format specified by FRF (e.g. Dual-SPI)		

SSI: TXD_DRIVE_EDGE Register

Offset: 0xf8

Description

TX drive edge

Table 610.
TXD_DRIVE_EDGE
Register

Bits	Description	Type	Reset
31:8	Reserved.	-	-
7:0	TDE : TXD drive edge	RW	0x00

表609。
SPI_CTRLR0寄存器

位	描述	类型	复位值
31:24	XIP_CMD: XIP模式下发送的SPI命令（INST_L = 8位）或附加至地址的命令（INST_L = 0位）	读写	0x03
23:19	保留。	-	-
18	SPI_RXDS_EN: 读数据触发使能	读写	0x0
17	INST_DDR_EN: 指令DDR传输使能	读写	0x0
16	SPI_DDR_EN: SPI DDR传输使能	读写	0x0
15:11	WAIT_CYCLES: 控制帧传输与数据接收之间的等待周期（SCLK周期）	读写	0x00
10	保留。	-	-
9:8	INST_L: 指令长度（0/4/8/16位）	读写	0x0
	枚举值：		
	0x0 → NONE：无指令		
	0x1 → 4B：4位指令		
	0x2 → 8B：8位指令		
	0x3 → 16B：16位指令		
7:6	保留。	-	-
5:2	ADDR_L: 地址长度（以4位递增，范围0b-60b）	读写	0x0
1:0	TRANS_TYPE: 地址与指令传输格式	读写	0x0
	枚举值：		
	0x0 → 1C1A：命令和地址均采用标准SPI帧格式		
	0x1 → 1C2A：命令采用标准SPI格式，地址采用指定格式由FRF指定		
	0x2 → 2C2A：命令和地址均采用FRF指定的格式（例如双SPI）		

SSI: TXD_DRIVE_EDGE寄存器

偏移: 0xf8

说明

TX驱动边沿

表610。
TXD_DRIVE_EDGE
寄存器

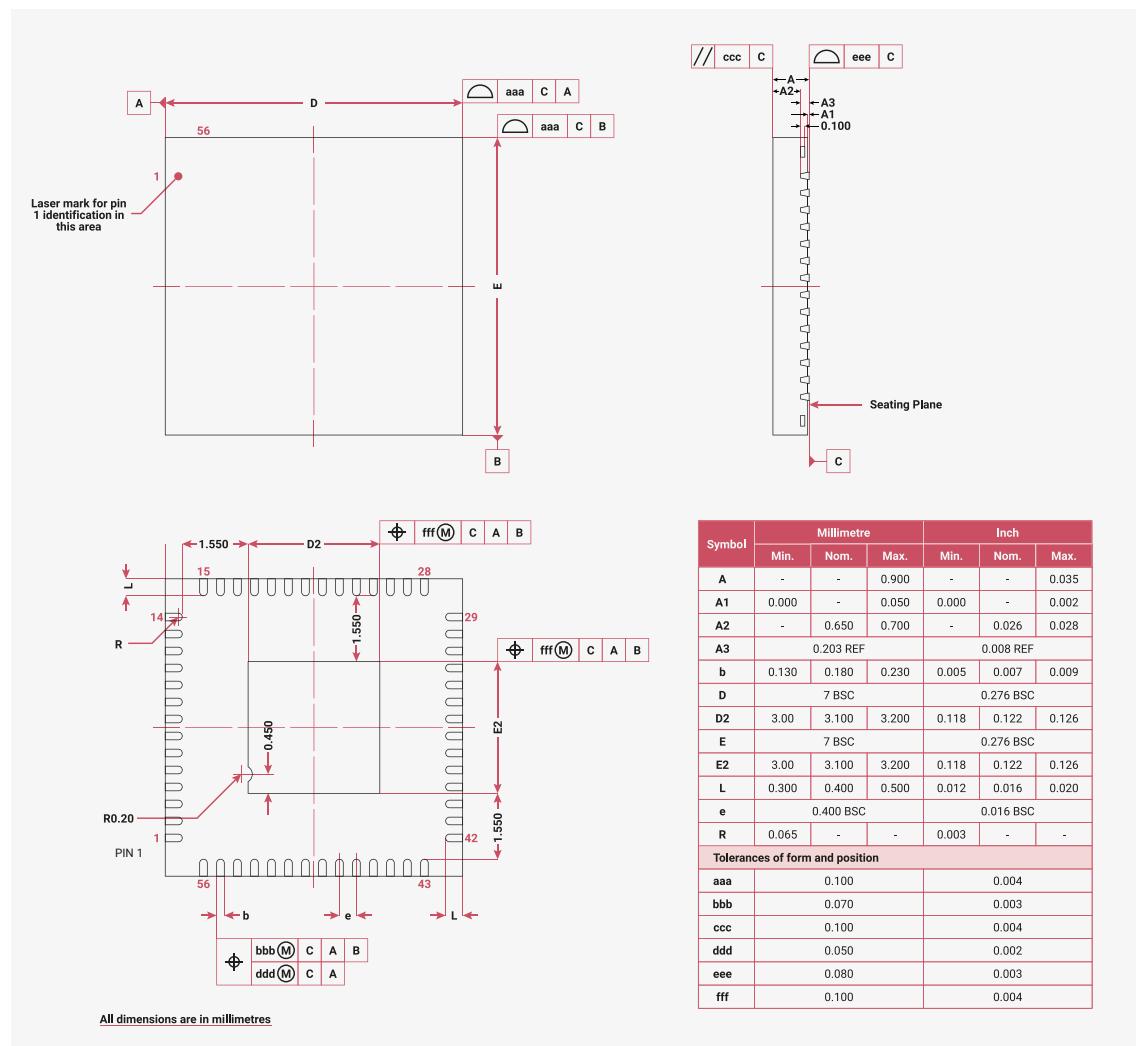
位	描述	类型	复位值
31:8	保留。	-	-
7:0	TDE: TXD 驱动边沿	读写	0x00

Chapter 5. Electrical and Mechanical

Physical and electrical details of the RP2040 chip.

5.1. Package

Figure 166. Top down view (left, top) and side view (right, top), along with bottom view (left, bottom) of the RP2040 QFN-56 package



NOTE

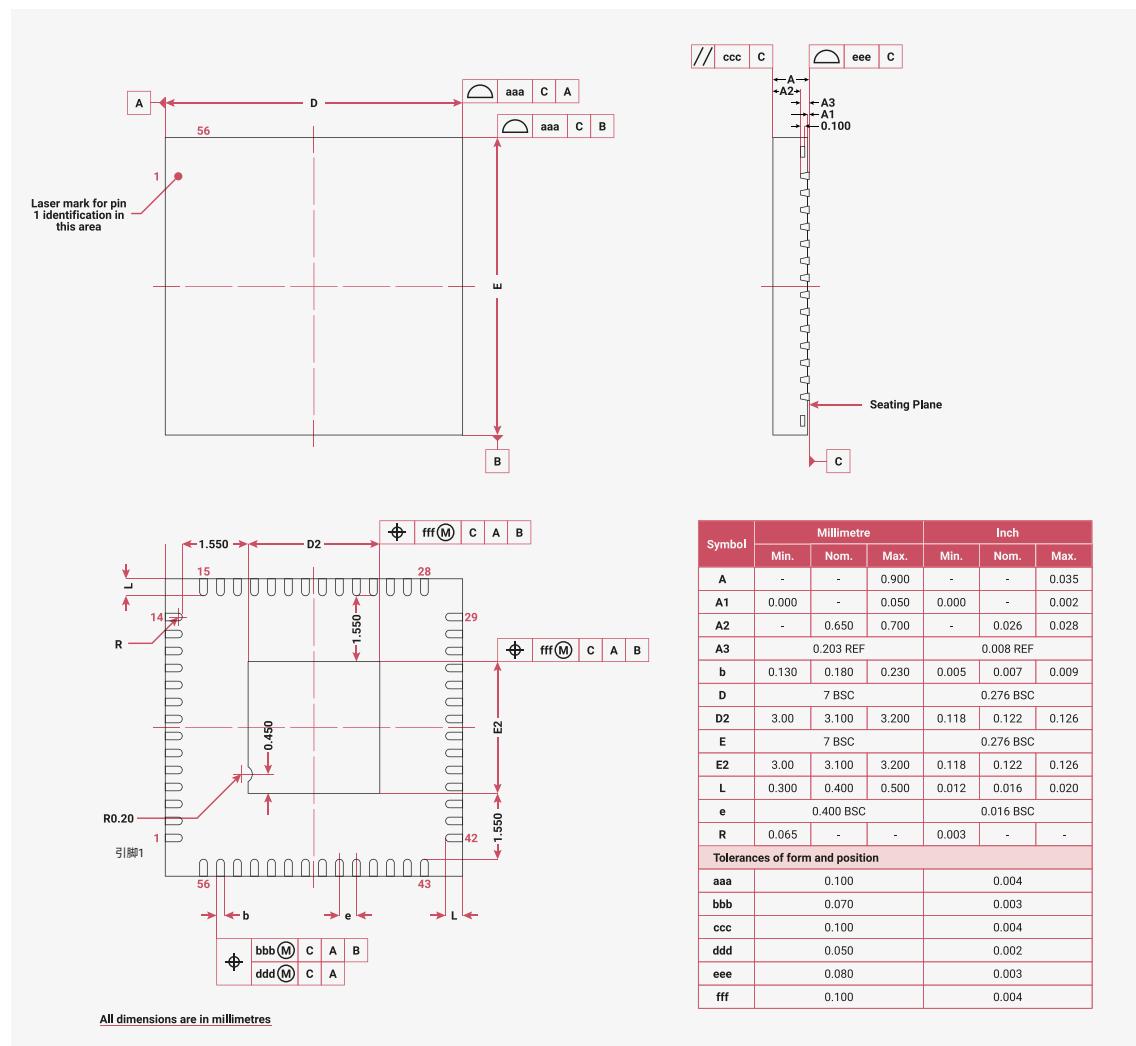
There is no standard size for the central GND pad (or ePad) with QFNs. However, the one on RP2040 is smaller than most. This means that standard 0.4mm QFN-56 footprints provided with CAD tools may need adjusting. This gives the opportunity to route between the central pad and the ones on the periphery, which can help with maintaining power and ground integrity on cheaper PCBs. See [Minimal Design Example](#) for an example.

第5章。电气与机械

RP2040芯片的物理与电气细节。

5.1. 封装

图166。RP2040 QFN-56封装的俯视图（左上）、侧视图（右上）及底视图（左下）



注意

QFN封装中的中央接地焊盘（或ePad）无统一标准尺寸，然而RP2040上的尺寸小于多数封装。这意味着附带CAD工具的标准0.4mm QFN-56焊盘图可能需进行调整。此设计允许在中央焊盘与外围焊盘间布线，有助于维护较低成本PCB上的电源及接地完整性。详见最小设计示例。

NOTE

Leads have a matte Tin (Sn) finish. Annealing is done post-plating, baking at 150°C for 1 hour. Minimum thickness for lead plating is 8 microms, and the intermediate layer material is CuFe2P (roughened Copper (Cu)).

5.1.1. Thermal characteristics

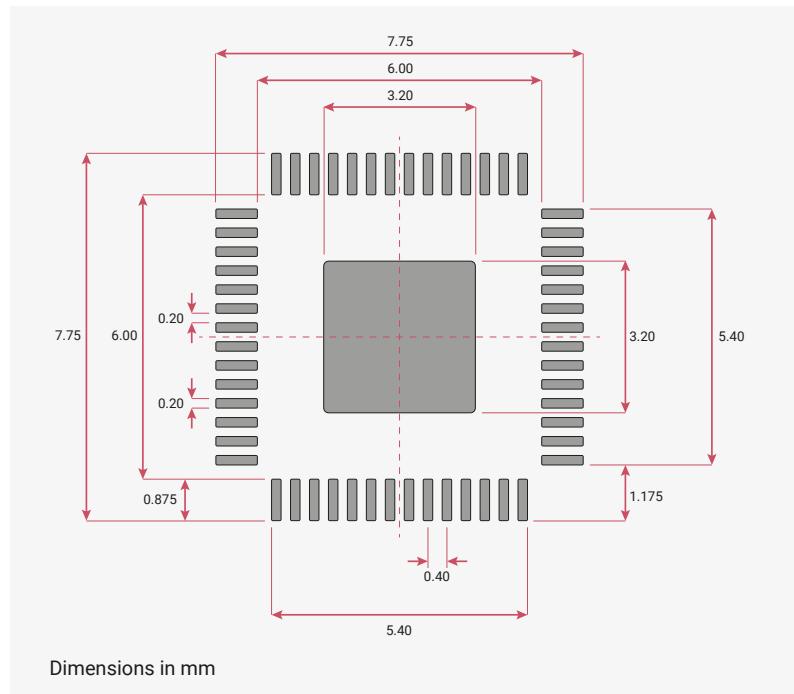
The thermal characteristics of the package are shown in [Table 611](#).

Table 611. Thermal data for the RP2040 QFN 56 package.

θ_{JA} (°C/W)	Ψ_{JT} (°C/W)	Ψ_{JB} (°C/W)	T_J (°C)	T_T (°C)	θ_{JC} (°C/W)	θ_{JB} (°C/W)
48.00	0.80	29.20	42.00	41.8	19.01	29.03

5.1.2. Recommended PCB Footprint

Figure 167. Recommended PCB Footprint for the RP2040 QFN-56 package



5.1.3. Package markings

The RP2040 7×7 mm QFN-56 package is marked as seen in [Figure 168](#), with specifications as shown in [Table 612](#). Coordinate origin is bottom-left of the package.

● 注意

引脚涂覆哑光锡（Sn）表面。退火在电镀后进行，条件为150°C，持续1小时。引脚镀层最低厚度为8微米，中间层材料为CuFe2P（粗化铜，Cu）。

5.1.1. 热特性

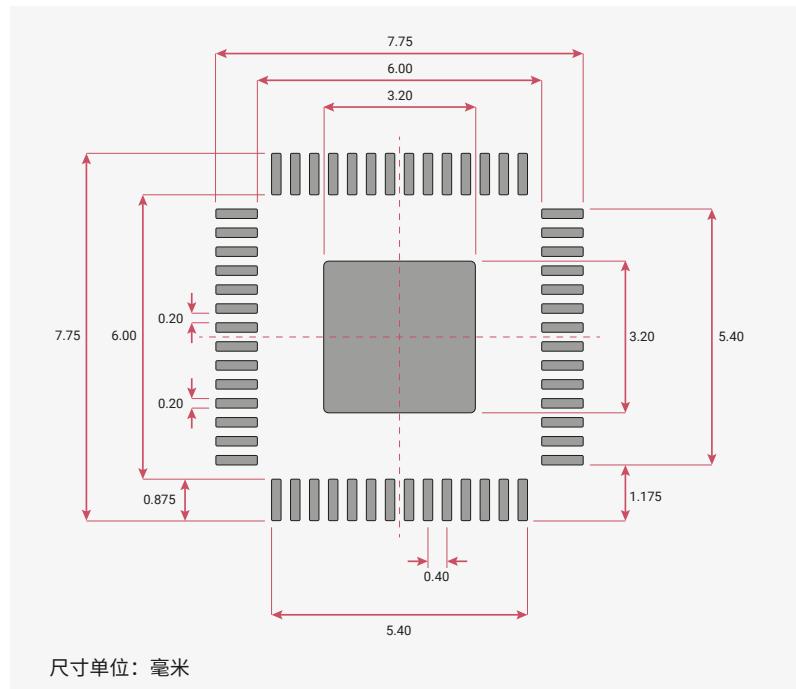
封装热性能数据详见表611。

表611。RP2040 QFN 56封装的热性能数据。

θ_{JA} (°C/W)	Ψ_{JT} (°C/W)	Ψ_{JB} (°C/W)	T_J (°C)	T_T (°C)	θ_{JC} (°C/W)	θ_{JB} (°C/W)
48.00	0.80	29.20	42.00	41.8	19.01	29.03

5.1.2. 推荐 PCB 布局

图 167。
RP2040 QFN-56 封装的推荐 PCB 封装尺寸



5.1.3. 封装标记

RP2040 7x7 毫米 QFN-56 封装如图 168 所示，规格见表 612。
坐标原点位于封装左下角。

Figure 168. Package marking format

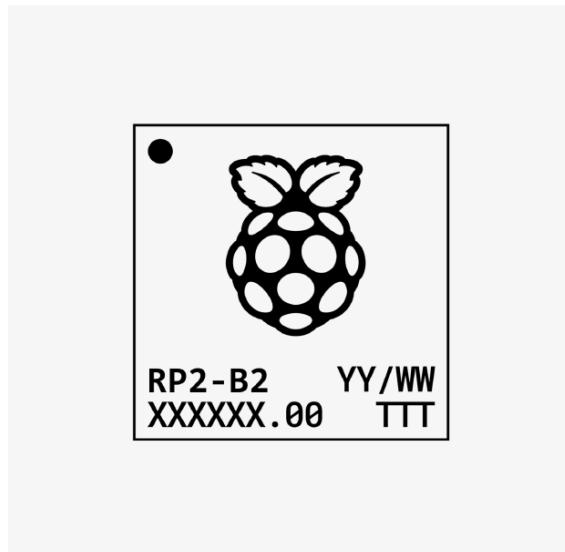


Table 612. Marking requirements and dimensions

Line	Step	Item	Coord. X	Coord. Y	Char. Height	Char. Width	Char. Space
1	1	Pin 1 Dot	0.5	6	0.5	0.5	
2	1	Logo	3.5	2.395	3.83	3.05	
3	1	RP2-B2	0.555	1.585	0.61	0.37	0.09
3	2	YY/WW	4.235	1.585	0.61	0.37	0.09
4	1	XXXXXX.00	0.555	0.775	0.61	0.37	0.09
4	2	TTT (optional)	5.155	0.775	0.61	0.37	0.09

NOTE

At Line 3, Step 1, the "RP2-B2" marking denotes device name "RP2" and silicon revision "B2."

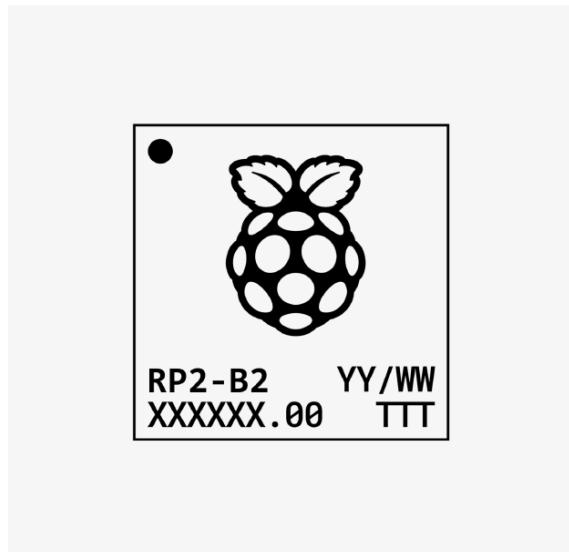
5.2. Storage conditions

In order to preserve the shelf and floor life of bare RP2040 devices, the recommended storage conditions in line with J-STD (020E & 033D) for RP2040 (classified MSL1) should be kept under 30°C and 85% relative humidity.

5.3. Solder profile

RP2040 is a Pb-free part, with a T_p value of 260°C.

All temperatures refer to the center of the package, measured on the package body surface that is facing up during assembly reflow (live-bug orientation). If parts are reflowed in other than the normal live-bug assembly reflow orientation (i.e., dead-bug), T_p shall be within $\pm 2^\circ\text{C}$ of the live-bug T_p and still meet the T_c requirements; otherwise, the profile shall be adjusted to achieve the latter.

图 168。封装
标记格式表 612。标记
要求及
尺寸

线	步骤	项目	X 坐标	坐标 Y	字符高度	字符宽度	字符间距
1	1	引脚1点	0.5	6	0.5	0.5	
2	1	标识	3.5	2.395	3.83	3.05	
3	1	RP2-B2	0.555	1.585	0.61	0.37	0.09
3	2	YY/WW	4.235	1.585	0.61	0.37	0.09
4	1	XXXXXX.00	0.555	0.775	0.61	0.37	0.09
4	2	TTT (可选)	5.155	0.775	0.61	0.37	0.09

① 注意

在第3行第1步中，“RP2-B2”标记表示器件名称“RP2”及硅片版本“B2”。

5.2. 存储条件

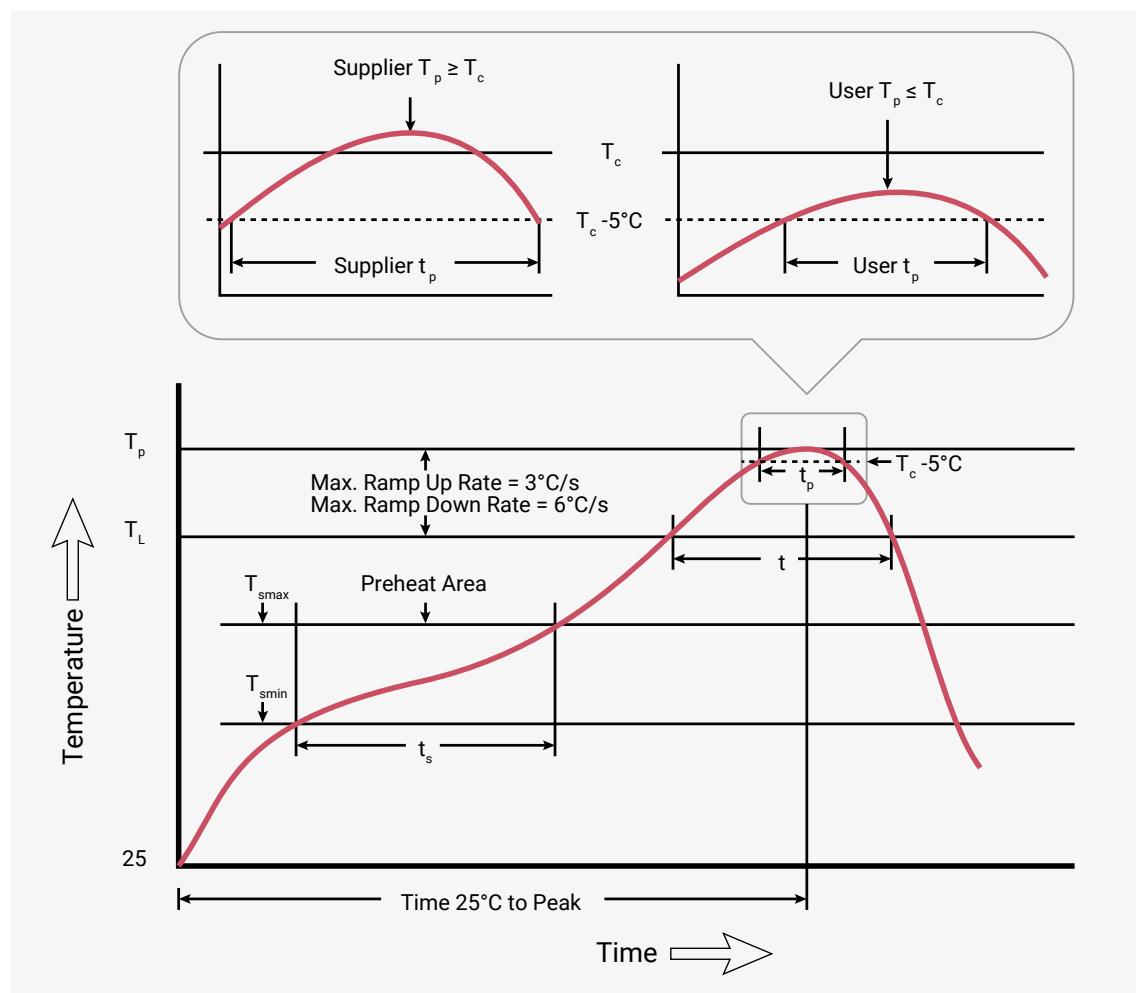
为保持裸露RP2040器件的货架期及存放寿命，建议依照J-STD（020E及033D）针对RP2040（归类为MSL1）的储存条件，保持温度低于30°C、相对湿度不超过85%。

5.3. 焊接曲线

RP2040为无铅器件，其 T_p 值为260°C。

所有温度均指封装中心，测量于组装返修时朝上的封装主体表面（live-bug方向）。若器件以非正常live-bug返修方向（即dead-bug）进行返修， T_p 应控制在live-bug $T_p \pm 2^\circ\text{C}$ 内，且仍需满足 T_c 要求；否则，应调整曲线以实现后者。

Figure 169.
Classification profile
(not to scale)



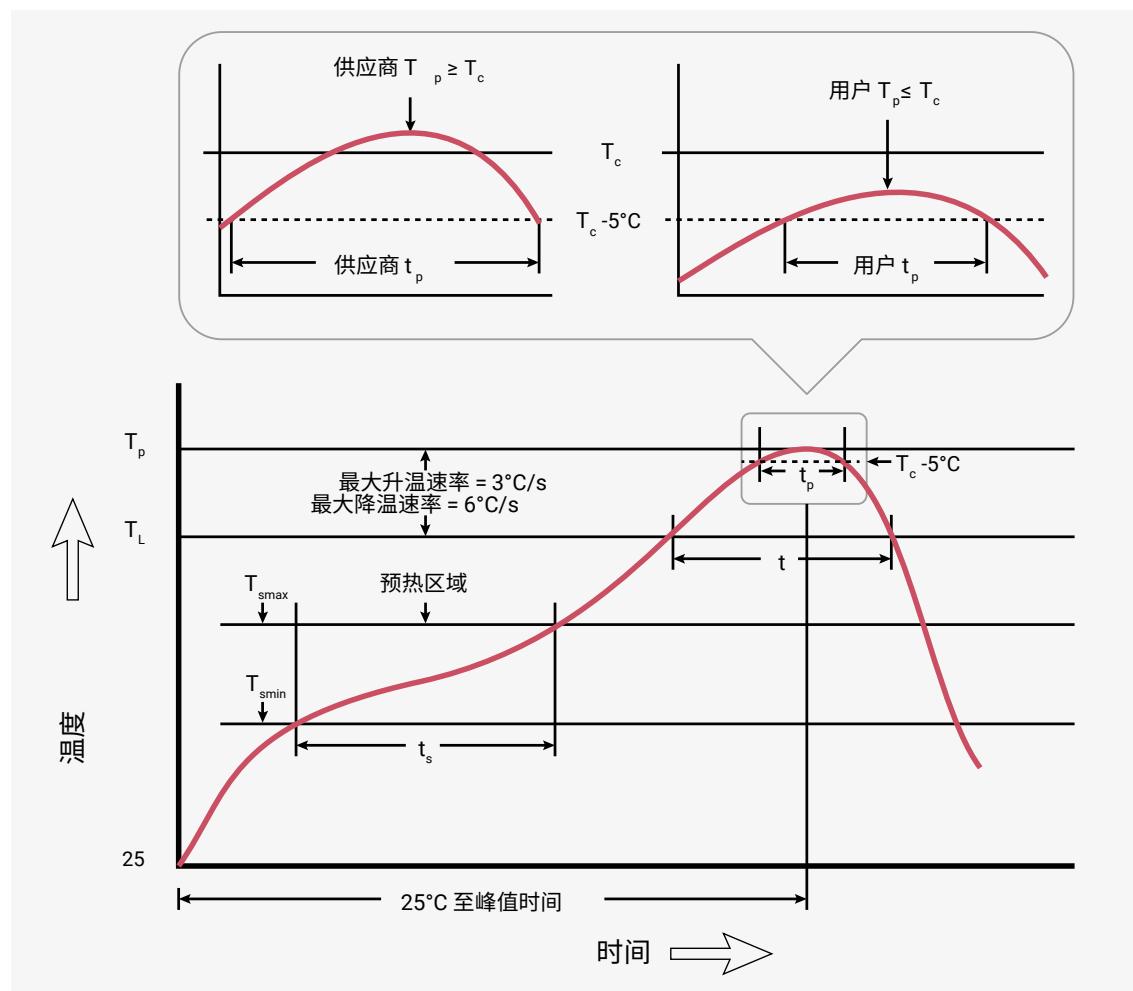
NOTE

Reflow profiles in this document are for classification/preconditioning, and are not meant to specify board assembly profiles. Actual board assembly profiles should be developed based on specific process needs and board designs, and should not exceed the parameters in [Table 613](#).

Table 613. Solder profile values

Profile feature	Value
Temperature min ($T_{s\min}$)	150°C
Temperature max ($T_{s\max}$)	200°C
Time (t_s) from ($T_{s\min}$ to $T_{s\max}$)	60 – 120 seconds
Ramp-up rate (T_L to T_p)	3°C/second max.
Liquidous temperature (T_L)	217°C
Time (t_L) maintained above T_L	60 to 150 seconds
Peak package body temperature (T_p)	260°C
Classification temperature (T_c)	260°C
Time (t_p) within 5°C of the specified classification temperature (T_c)	30 seconds
Ramp-down rate (T_p to T_L)	6°C/second max.
Time 25°C to peak temperature	8 minutes max.

图169。
分类轮廓
(非按比例绘制)



① 注意

本文件中的回流轮廓用于分类/预处理，不应用于规定电路板组装轮廓。实际电路板组装轮廓应依据具体工艺需求和电路板设计制定，且不得超过表613中的参数。

表613。焊接轮廓数值

轮廓特征	数值
最低温度 (T_{smin})	150°C
最高温度 (T_{smax})	200°C
从 (T_{smin} 至 T_{smax}) 的时间 (t_s)	60–120秒
升温速率 (从 T_L 至 T_p)	最大 3°C/s
液相温度 (T_L)	217°C
保持高于 T_L 的时间 (t_L)	60 到 150 秒
峰值封装体温 (T_p)	260°C
分类温度 (T_c)	260°C
在规定分类温度 (T_c) $\pm 5^\circ\text{C}$ 范围内的时间 (t_p)	30 秒
降温速率 (从 T_p 到 T_L)	最大 6°C/s
从 25°C 到峰值温度的时间	最长 8 分钟

5.4. Compliance

RP2040 is compliant to Moisture Sensitivity Level 1.

RP2040 is compliant to the requirement of REACH Substances of Very High Concern (SVHC) that ECHA announced on 25 June 2020.

RP2040 is compliant to the requirement and standard of Controlled Environment-related Substance of RoHS directive (EU) 2011/65/EU and directive (EU) 2015/863.

Package Level reliability qualifications carried out on RP2040:

- Temperature Cycling per JESD22-A104
- HAST per JESD22-A110
- HTSL per JESD22-A103

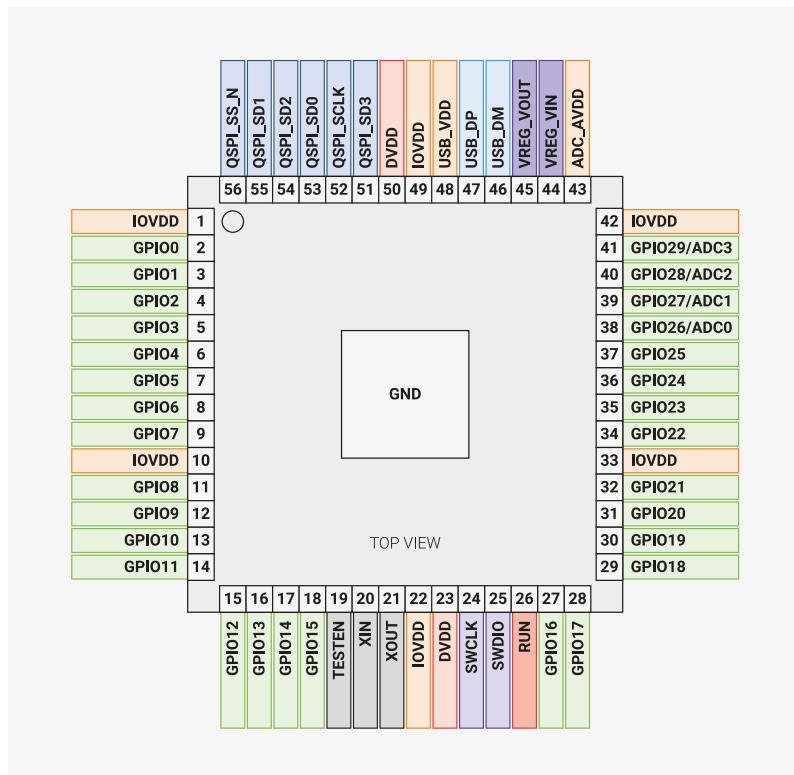
i NOTE

A tin whiskers test is not performed as RP2040 is a bottom only termination device (QFN package) which not applicable to JEDEC standard (JESD201A).

5.5. Pinout

5.5.1. Pin Locations

Figure 170. RP2040 QFN-56 package pinout



5.4. 合规性

RP2040 符合湿度敏感等级 1 的要求。

RP2040 符合欧洲化学品管理署（ECHA）于2020年6月25日公布的高度关注物质（SVHC）REACH要求。

RP2040 符合RoHS指令（欧盟）2011/65/EU及指令（欧盟）2015/863中有关受控环境相关物质的标准及要求。

RP2040所进行的封装级可靠性鉴定：

- 温度循环测试，依据 JESD22-A104 标准
- 高加速应力测试（HAST），依据 JESD22-A110 标准
- 高温储存寿命测试（HTSL），依据 JESD22-A103 标准

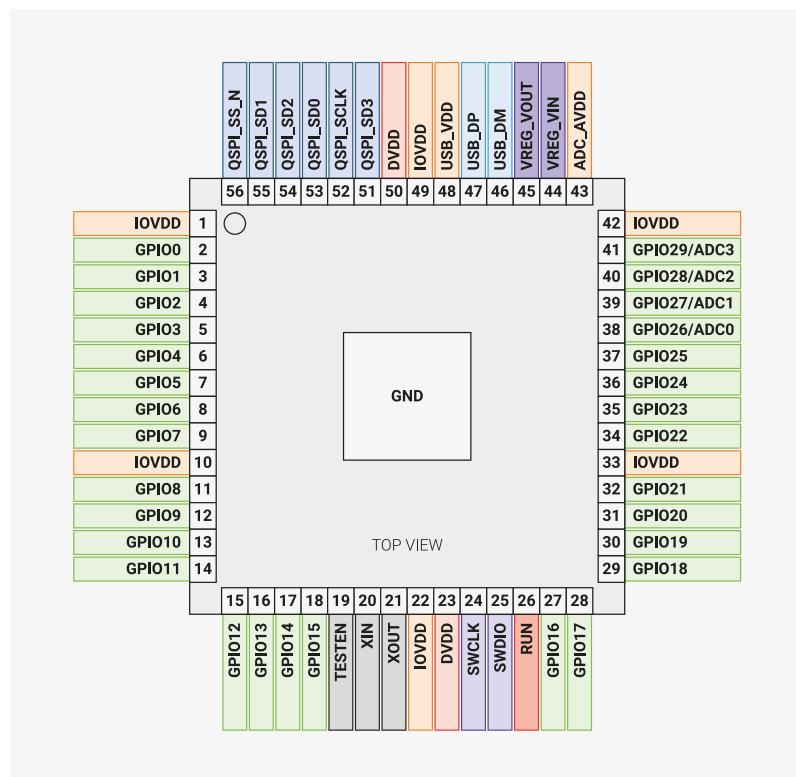
i 注意

由于RP2040为底部终端器件（QFN封装），不适用JEDEC标准（JESD201A），因此未进行锡须测试。

5.5. 引脚定义

5.5.1. 引脚位置

图170. RP2040
QFN-56封装
引脚排列



5.5.2. Pin Definitions

5.5.2.1. Pin Types

In the following GPIO Pin table (Table 615), the pin types are defined as shown below.

Table 614. Pin Types

Pin Type	Direction	Description
Digital In	Input only	Standard Digital. Programmable Pull-Up, Pull-Down, Slew Rate, Schmitt Trigger and Drive Strength. Default Drive Strength is 4mA.
Digital IO	Bi-directional	
Digital In (FT)	Input only	Fault Tolerant Digital. These pins are described as Fault Tolerant, which in this case means that very little current flows into the pin whilst it is below 3.63V and IOVDD is 0V. There is also enhanced ESD protection on these pins. Programmable Pull-Up, Pull-Down, Slew Rate, Schmitt Trigger and Drive Strength. Default Drive Strength is 4mA.
Digital IO (FT)	Bi-directional	
Digital IO / Analogue	Bi-directional (digital), Input (Analogue)	Standard Digital and ADC input. Programmable Pull-Up, Pull-Down, Slew Rate, Schmitt Trigger and Drive Strength. Default Drive Strength is 4mA.
USB IO	Bi-directional	These pins are for USB use, and contain internal pull-up and pull-down resistors, as per the USB specification. Note that external 27Ω series resistors are required for USB operation.
Analogue (XOSC)		Oscillator input pins for attaching a 12MHz crystal. Alternatively, XIN may be driven by a square wave.

5.5.2.2. Pin List

Table 615. GPIO pins

Name	Number	Type	Power Domain	Reset State	Description
GPIO0	2	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO1	3	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO2	4	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO3	5	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO4	6	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO5	7	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO6	8	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO7	9	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO8	11	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO9	12	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO10	13	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO11	14	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO12	15	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO13	16	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO14	17	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO15	18	Digital IO (FT)	IOVDD	Pull-Down	User IO

5.5.2. 引脚说明

5.5.2.1. 引脚类型

下表GPIO引脚（表615）中，引脚类型定义如下。

表614. 引脚类型

引脚类型	方向	描述
数字输入	仅输入	标准数字。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
数字输入输出	双向	
数字输入（容错）	仅输入	容错数字。这些引脚被描述为容错型，即当引脚电压低于3.63V且IOVDD为0V时，流入该引脚的电流极小。这些引脚还具备增强的ESD保护功能。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
数字IO (FT)	双向	
数字IO / 模拟	双向（数字），输入（模拟）	标准数字及ADC输入。可编程上拉、下拉、转换速率、施密特触发和驱动强度。默认驱动强度为4mA。
USB IO	双向	这些引脚用于USB功能，依据USB规范内含内部上拉及下拉电阻。注意USB操作需外部27Ω串联电阻。
模拟(XOSC)		用于连接12MHz晶体的振荡器输入引脚。或者，XIN端可由方波信号驱动。

5.5.2.2. 引脚列表

表615. GPIO引脚

名称	编号	类型	电源域	复位状态	描述
GPIO0	2	数字IO (FT)	IOVDD	下拉	用户IO
GPIO1	3	数字IO (FT)	IOVDD	下拉	用户IO
GPIO2	4	数字IO (FT)	IOVDD	下拉	用户IO
GPIO3	5	数字IO (FT)	IOVDD	下拉	用户IO
GPIO4	6	数字IO (FT)	IOVDD	下拉	用户IO
GPIO5	7	数字IO (FT)	IOVDD	下拉	用户IO
GPIO6	8	数字IO (FT)	IOVDD	下拉	用户IO
GPIO7	9	数字IO (FT)	IOVDD	下拉	用户IO
GPIO8	11	数字IO (FT)	IOVDD	下拉	用户IO
GPIO9	12	数字IO (FT)	IOVDD	下拉	用户IO
GPIO10	13	数字IO (FT)	IOVDD	下拉	用户IO
GPIO11	14	数字IO (FT)	IOVDD	下拉	用户IO
GPIO12	15	数字IO (FT)	IOVDD	下拉	用户IO
GPIO13	16	数字IO (FT)	IOVDD	下拉	用户IO
GPIO14	17	数字IO (FT)	IOVDD	下拉	用户IO
GPIO15	18	数字IO (FT)	IOVDD	下拉	用户IO

Name	Number	Type	Power Domain	Reset State	Description
GPIO16	27	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO17	28	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO18	29	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO19	30	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO20	31	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO21	32	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO22	34	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO23	35	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO24	36	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO25	37	Digital IO (FT)	IOVDD	Pull-Down	User IO
GPIO26 / ADC0	38	Digital IO / Analogue	IOVDD / ADC_AVDD	Pull-Down	User IO or ADC input
GPIO27 / ADC1	39	Digital IO / Analogue	IOVDD / ADC_AVDD	Pull-Down	User IO or ADC input
GPIO28 / ADC2	40	Digital IO / Analogue	IOVDD / ADC_AVDD	Pull-Down	User IO or ADC input
GPIO29 / ADC3	41	Digital IO / Analogue	IOVDD / ADC_AVDD	Pull-Down	User IO or ADC input

Table 616. QSPI pins

Name	Number	Type	Power Domain	Reset State	Description
QSPI_SD3	51	Digital IO	IOVDD		QSPI data
QSPI_SCLK	52	Digital IO	IOVDD	Pull-Down	QSPI clock
QSPI_SD0	53	Digital IO	IOVDD		QSPI data
QSPI_SD2	54	Digital IO	IOVDD		QSPI data
QSPI_SD1	55	Digital IO	IOVDD		QSPI data
QSPI_CSn	56	Digital IO	IOVDD	Pull-Up	QSPI chip select

Table 617. Crystal oscillator pins

Name	Number	Type	Power Domain	Description
XIN	20	Analogue (XOSC)	IOVDD	Crystal oscillator. XIN may also be driven by a square wave.
XOUT	21	Analogue (XOSC)	IOVDD	Crystal oscillator.

Table 618. Serial wire debug pins

Name	Number	Type	Power Domain	Reset State	Description
SWCLK	24	Digital In (FT)	IOVDD	Pull-Up	Debug clock
SWD	25	Digital IO (FT)	IOVDD	Pull-Up	Debug data

Table 619. Miscellaneous pins

Name	Number	Type	Power Domain	Reset State	Description
RUN	26	Digital In (FT)	IOVDD	Pull-Up	Chip enable / reset

名称	编号	类型	电源域	复位状态	描述
GPIO16	27	数字IO (FT)	IOVDD	下拉	用户IO
GPIO17	28	数字IO (FT)	IOVDD	下拉	用户IO
GPIO18	29	数字IO (FT)	IOVDD	下拉	用户IO
GPIO19	30	数字IO (FT)	IOVDD	下拉	用户IO
GPIO20	31	数字IO (FT)	IOVDD	下拉	用户IO
GPIO21	32	数字IO (FT)	IOVDD	下拉	用户IO
GPIO22	34	数字IO (FT)	IOVDD	下拉	用户IO
GPIO23	35	数字IO (FT)	IOVDD	下拉	用户IO
GPIO24	36	数字IO (FT)	IOVDD	下拉	用户IO
GPIO25	37	数字IO (FT)	IOVDD	下拉	用户IO
GPIO26 / ADC0	38	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO27 / ADC1	39	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO28 / ADC2	40	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入
GPIO29 / ADC3	41	数字 IO / 模拟	IOVDD / ADC_AVDD	下拉	用户 IO 或 ADC 输入

表 616. QSPI 引脚

名称	编号	类型	电源域	复位状态	描述
QSPI_SD3	51	数字输入输出	IOVDD		QSPI 数据
QSPI_SCLK	52	数字输入输出	IOVDD	下拉	QSPI 时钟
QSPI_SD0	53	数字输入输出	IOVDD		QSPI 数据
QSPI_SD2	54	数字输入输出	IOVDD		QSPI 数据
QSPI_SD1	55	数字输入输出	IOVDD		QSPI 数据
QSPI_CS _n	56	数字输入输出	IOVDD	上拉	QSPI 片选

表 617. 晶体振荡器引脚

名称	编号	类型	电源域	描述
XIN	20	模拟 (XOSC)	IOVDD	晶体振荡器。XIN 也可由方波驱动。
XOUT	21	模拟 (XOSC)	IOVDD	晶体振荡器。

表 618. 串行线调试引脚

名称	编号	类型	电源域	复位状态	描述
SWCLK	24	数字输入 (容错)	IOVDD	上拉	调试时钟
SWD	25	数字IO (FT)	IOVDD	上拉	调试数据

表 619. 杂项引脚

名称	编号	类型	电源域	复位状态	描述
RUN	26	数字输入 (容错)	IOVDD	上拉	芯片使能 / 复位

Name	Number	Type	Power Domain	Reset State	Description
TESTEN	19	Digital In	IOVDD	Pull-Down	Test enable (connect to Gnd)

Table 620. USB pins

Name	Number	Type	Power Domain	Description
USB_DP	47	USB IO	USB_VDD	USB Data +ve. 27Ω series resistor required for USB operation
USB_DM	46	USB IO	USB_VDD	USB Data -ve. 27Ω series resistor required for USB operation

Table 621. Power supply pins

Name	Number(s)	Description
IOVDD	1, 10, 22, 33, 42, 49	IO supply
DVDD	23, 50	Core supply
VREG_VIN	44	Voltage regulator input supply
VREG_VOUT	45	Voltage regulator output
USB_VDD	48	USB supply
ADC_AVDD	43	ADC supply
GND	57	Common ground connection via central pad

5.5.3. Pin Specifications

The following electrical specifications are obtained from characterisation over the specified temperature and voltage ranges, as well as process variation, unless the specification is marked as 'Simulated'. In this case, the data is for information purposes only, and is not guaranteed.

5.5.3.1. Absolute Maximum Ratings

Table 622. Absolute maximum ratings for digital IO (Standard and Fault Tolerant)

Parameter	Symbol	Minimum	Maximum	Units	Comment
I/O Supply Voltage	IOVDD	-0.5	3.63	V	
Voltage at IO	V_{PIN}	-0.5	IOVDD + 0.5	V	

5.5.3.2. ESD Performance

Table 623. ESD performance for all pins, unless otherwise stated

Parameter	Symbol	Maximum	Units	Comment
Human Body Model	HBM	2	kV	Compliant with JEDEC specification JS-001-2012 (April 2012)

名称	编号	类型	电源域	复位状态	描述
测试	19	数字输入	IOVDD	下拉	测试使能（连接至接地）

表 620. USB 引脚

名称	编号	类型	电源域	描述
USB_DP	47	USB IO	USB_VDD	USB 数据正极，USB 运行需串联27Ω电阻
USB_DM	46	USB IO	USB_VDD	USB 数据负极，USB 运行需串联27Ω电阻

表 621. 电源引脚

名称	编号	描述
IOVDD	1, 10, 22, 33, 42, 49	IO 电源
DVDD	23, 50	核心电源
VREG_VIN	44	稳压器输入电源
VREG_VOUT	45	稳压器输出
USB_VDD	48	USB 电源
ADC_AVDD	43	ADC 电源
GND	57	通过中心焊盘连接公共地线

5.5.3. 引脚规格

以下电气规格基于指定温度和电压范围内的特性测试及工艺变异获得，除非规格标注为“模拟”。在此情况下，数据仅供参考，且不保证其准确性。

5.5.3.1. 绝对最大额定值

表 622。 数字IO（标准与容错）绝对最大额定值

参数	符号	最小值	最大值	单位	备注
I/O 供电电压	IOVDD	-0.5	3.63	V	
I/O端电压	V _{引脚}	-0.5	IOVDD + 0.5	V	

5.5.3.2. 静电放电性能

表 623。 除非另有说明，所有引脚的静电放电性能

参数	符号	最大值	单位	备注
人体模型	HBM	2	kV	符合 JEDEC 规格 JS-01-2012 (2012年4月)

Parameter	Symbol	Maximum	Units	Comment
<i>Human Body Model Digital (FT) pins only</i>	HBM	4	kV	Compliant with JEDEC specification JS-001-2012 (April 2012)
<i>Charged Device Model</i>	CDM	500	V	Compliant with JESD22-C101E (December 2009)

5.5.3.3. Thermal Performance

Table 624. Thermal Performance

Parameter	Symbol	Minimum	Typical	Maximum	Units	Comment
Case Temperature	T _C	-40		85	°C	

5.5.3.4. IO Electrical Characteristics

Table 625. Digital IO characteristics - Standard and FT unless otherwise stated

Parameter	Symbol	Minimum	Maximum	Units	Comment
Pin Input Leakage Current	I _{IN}		1	µA	
Input Voltage High @ IOVDD=1.8V	V _{IH}	0.65 * IOVDD	IOVDD + 0.3	V	
Input Voltage High @ IOVDD=2.5V	V _{IH}	1.7	IOVDD + 0.3	V	
Input Voltage High @ IOVDD=3.3V	V _{IH}	2	IOVDD + 0.3	V	
Input Voltage Low @ IOVDD=1.8V	V _{IL}	-0.3	0.35 * IOVDD	V	
Input Voltage Low @ IOVDD=2.5V	V _{IL}	-0.3	0.7	V	
Input Voltage Low @ IOVDD=3.3V	V _{IL}	-0.3	0.8	V	
Input Hysteresis Voltage @ IOVDD=1.8V	V _{HYS}	0.1 * IOVDD		V	Schmitt Trigger enabled
Input Hysteresis Voltage @ IOVDD=2.5V	V _{HYS}	0.2		V	Schmitt Trigger enabled
Input Hysteresis Voltage @ IOVDD=3.3V	V _{HYS}	0.2		V	Schmitt Trigger enabled
Output Voltage High @ IOVDD=1.8V	V _{OH}	1.24	IOVDD	V	I _{OH} = 2, 4, 8 or 12mA depending on setting

参数	符号	最大值	单位	备注
人体模型 仅适用于数字 (FT) 引脚	HBM	4	kV	符合 JEDEC 规格 JS-0 01-2012 (2012年4 月)
带电器件模型	CDM	500	V	符合 JESD22-C101E (2009年12月)

5.5.3.3. 热性能

表624. 热性能

参数	符号	最小值	典型值	最大值	单位	备注
封装 温度	T _C	-40		85	°C	

5.5.3.4. IO电气特性

表625. 数字IO特性
- 标准及FT,
除非另有说明

参数	符号	最小值	最大值	单位	备注
引脚输入漏电流	I _{IN}		1	μA	
输入高电压 @ IOVDD=1.8V	V _{IH}	0.65 * IOVDD	IOVDD + 0.3	V	
输入高电压 @ IOVDD=2.5V	V _{IH}	1.7	IOVDD + 0.3	V	
输入高电压 @ IOVDD=3.3V	V _{IH}	2	IOVDD + 0.3	V	
输入电压低 @ IOVDD=1.8V	V _{IL}	-0.3	0.35 * IOVDD	V	
输入电压低 @ IOVDD=2.5V	V _{IL}	-0.3	0.7	V	
输入电压低 @ IOVDD=3.3V	V _{IL}	-0.3	0.8	V	
输入迟滞 电压 @ IOVDD=1.8V	V _{HYS}	0.1 * IOVDD		V	施密特触发器 已启用
输入迟滞 电压 @ IOVDD=2.5V	V _{HYS}	0.2		V	施密特触发器 已启用
输入迟滞 电压 @ IOVDD=3.3V	V _{HYS}	0.2		V	施密特触发器 已启用
输出电压 高电平 @ IOVDD=1.8V	V _{OH}	1.24	IOVDD	V	I _{OH} = 2、4、8 或12mA，取决 于设置

Parameter	Symbol	Minimum	Maximum	Units	Comment
Output Voltage High @ IOVDD=2.5V	V _{OH}	1.78	IOVDD	V	I _{OH} = 2, 4, 8 or 12mA depending on setting
Output Voltage High @ IOVDD=3.3V	V _{OH}	2.62	IOVDD	V	I _{OH} = 2, 4, 8 or 12mA depending on setting
Output Voltage Low @ IOVDD=1.8V	V _{OL}	0	0.3	V	I _{OL} = 2, 4, 8 or 12mA depending on setting
Output Voltage Low @ IOVDD=2.5V	V _{OL}	0	0.4	V	I _{OL} = 2, 4, 8 or 12mA depending on setting
Output Voltage Low @ IOVDD=3.3V	V _{OL}	0	0.5	V	I _{OL} = 2, 4, 8 or 12mA depending on setting
Pull-Up Resistance	R _{PU}	50	80	kΩ	
Pull-Down Resistance	R _{PD}	50	80	kΩ	
Maximum Total IOVDD current	I _{IOVDD_MAX}		50	mA	Sum of all current being sourced by GPIO and QSPI pins
Maximum Total VSS current due to IO (IOVSS)	I _{IOVSS_MAX}		50	mA	Sum of all current being sunk into GPIO and QSPI pins

Table 626. USB IO characteristics

Parameter	Symbol	Minimum	Maximum	Units	Comment
Pin Input Leakage Current	I _{IN}		1	µA	
Single Ended Input Voltage High	V _{IHSE}	2		V	
Single Ended Input Voltage Low	V _{ILSE}		0.8	V	
Differential Input Voltage High	V _{IHDIFF}	0.2		V	
Differential Input Voltage Low	V _{ILDIFF}		-0.2	V	
Output Voltage High	V _{OH}	2.8	USB_VDD	V	
Output Voltage Low	V _{OL}	0	0.3	V	
Pull-Up Resistance - R _{PU2}	R _{PU2}	0.873	1.548	kΩ	

参数	符号	最小值	最大值	单位	备注
输出电压 高电平 @ $I_{OVDD}=2.5V$	V_{OH}	1.78	I_{OVDD}	V	$I_{OH}=2、4、8$ 或12mA，取决于设置
输出电压 高电平 @ $I_{OVDD}=3.3V$	V_{OH}	2.62	I_{OVDD}	V	$I_{OH}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=1.8V$	V_{OL}	0	0.3	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=2.5V$	V_{OL}	0	0.4	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
输出电压 低电平 @ $I_{OVDD}=3.3V$	V_{OL}	0	0.5	V	$I_{OL}=2、4、8$ 或12mA，取决于设置
上拉电阻	R_{PU}	50	80	kΩ	
下拉 电阻	R_{PD}	50	80	kΩ	
最大总 I_{OVDD} 电流	$I_{I_{OVDD_MAX}}$		50	mA	GPIO 与 QSPI 引脚所提供电流之总和
最大总 由 $IO (I_{OVSS})$ 引起的 VSS 电流	$I_{I_{OVSS_MAX}}$		50	mA	流入 GPIO 与 QSPI 引脚的电流总和

表 626. USB IO 特性

参数	符号	最小值	最大值	单位	备注
引脚输入漏电流	I_{IN}		1	μA	
单端输入 高电压	V_{IHSE}	2		V	
单端输入 低电压	V_{ILSE}		0.8	V	
差分输入 高电压	V_{IHDIFF}	0.2		V	
差分输入 低电压	V_{ILDIFF}		-0.2	V	
输出电压 高	V_{OH}	2.8	USB_VDD	V	
输出电压 低	V_{OL}	0	0.3	V	
上拉电阻 - R_{PU2}	R_{PU2}	0.873	1.548	kΩ	

Parameter	Symbol	Minimum	Maximum	Units	Comment
Pull-Up Resistance - RPU1&2	R _{PU1&2}	1.398	3.063	kΩ	
Pull-Down Resistance	R _{PD}	14.25	15.75	kΩ	

Table 627. ADC characteristics

Parameter	Symbol	Minimum	Maximum	Units	Comment
ADC Input Voltage Range	V _{PIN_ADC}	0	ADC_AVDD	V	
Effective Number of Bits	ENOB	8.7		bits	See Section 4.9.3
Resolved Bits			12	bits	
ADC Input Impedance	R _{IN_ADC}	100		kΩ	

Table 628. Oscillator pin characteristics when using a Square Wave input

Parameter	Symbol	Minimum	Maximum	Units	Comment
Input Voltage High	V _{IH}	0.65*IOVDD	IOVDD + 0.3	V	XIN only. XOUT floating
Input Voltage Low	V _{IL}	0	0.35*IOVDD	V	XIN only. XOUT floating

See [Section 2.16](#) for more details on the Oscillator, and [Minimal Design Example](#) for information on crystal usage.

5.5.3.5. Interpreting GPIO output voltage specifications

The GPIOs on RP2040 have four different output drive strengths, which are nominally called 2, 4, 8 and 12mA modes. These are not hard limits, nor do they mean that they will always be sourcing (or sinking) the selected amount of milliamps. The amount of current a GPIO sources or sinks is dependent on the load attached to it. It will attempt to drive the output to the IOVDD level (or 0V in the case of a logic 0), but the amount of current it is able to source is limited, which will be dependent on the selected drive strength. Therefore the higher the current load is, the lower the voltage will be at the pin. At some point, the GPIO will be sourcing so much current, that the voltage is so low, it won't be recognised as a logic 1 by the input of a connected device. The purpose of the output specifications in [Table 625](#) are to try and quantify how much lower the voltage can be expected to be, when drawing specified amounts of current from the pin.

The Output High Voltage (V_{OH}) is defined as the lowest voltage the output pin can be when driven to a logic 1 with a particular selected drive strength; e.g., 4mA being sourced by the pin whilst in 4mA drive strength mode. The Output Low Voltage is similar, but with a logic 0 being driven.

In addition to this, the sum of all the IO currents being sourced (i.e. when outputs are being driven high) from the IOVDD bank (essentially the GPIO and QSPI pins), must not exceed I_{IOVDD_MAX}. Similarly, the sum of all the IO currents being sunk (i.e. when the outputs are being driven low) must not exceed I_{IOVSS_MAX}.

参数	符号	最小值	最大值	单位	备注
上拉电阻 — R _{PU1&2}	R _{PU1&2}	1.398	3.063	kΩ	
下拉 电阻	R _{PD}	14.25	15.75	kΩ	

表627. ADC
特性

参数	符号	最小值	最大值	单位	备注
ADC输入电压 范围	V _{PIN_ADC}	0	ADC_AVDD	V	
有效位数	ENOB	8.7		位	参见第4.9.3节
分辨位			12	位	
ADC输入 阻抗	R _{IN_ADC}	100		kΩ	

表628。使用方波
输入时的振荡器
引脚特性

参数	符号	最小值	最大值	单位	备注
输入高电压	V _{IH}	0.65*IOVDD	IOVDD + 0.3	V	仅XIN。XOUT 悬空
输入电压低	V _{IL}	0	0.35*IOVDD	V	仅XIN。XOUT 悬空

有关振荡器的详细信息，请参见第2.16节；有关晶体使用的信息，请参见最小设计示例。

5.5.3.5. GPIO输出电压规格的解释

RP2040上的GPIO具有四种不同的输出驱动强度，名义上分别称为2、4、8和12mA模式。

这并非硬性限制，也不意味着GPIO始终会提供（或吸收）所选的电流强度。GPIO提供或吸收的电流大小取决于所连接的负载。GPIO会尝试将输出驱动至IOVDD电平（逻辑0时为0V），但其可提供的电流受所选驱动强度限制。因此，电流负载越大，针脚处的电压越低。在某个临界点，GPIO输出的电流将非常大，导致电压降至无法被连接设备输入识别为逻辑1。表625中的输出规格旨在量化在从针脚拉取规定电流时，电压可能降低的幅度。

输出高电压（V_{OH}）定义为在特定选定驱动强度下，输出针脚驱动至逻辑1时的最低电压；例如，在4mA驱动强度模式下，针脚源出4mA电流。输出低电压的定义类似，但指针脚驱动逻辑0时的最低电压。

此外，从IOVDD电源组（即GPIO和QSPI针脚）输出的所有IO电流总和（当输出为高电平时）不得超过I_{IOVDD_MAX}。同样，所有被下拉吸收的IO电流总和（即输出被拉低时）不得超过I_{IOVSS_MAX}。

Figure 171. Typical Current vs Voltage curves of a GPIO output.

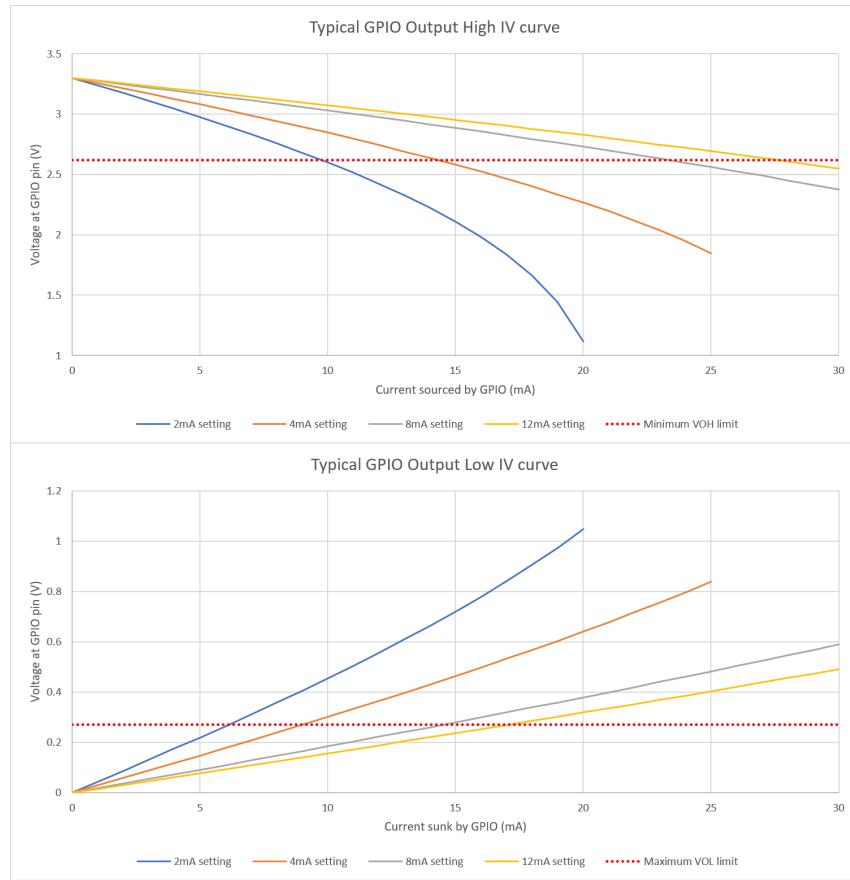


Figure 171 shows the effect on the output voltage as the current load on the pin increases. You can clearly see the effect of the different drive strengths; the higher the drive strength, the closer the output voltage is to IOVDD (or 0V) for a given current. The minimum V_{OH} and maximum V_{OL} limits are shown in red. You can see that at the specified current for each drive strength, the voltage is well within the allowed limits, meaning that this particular device could drive a lot more current and still be within V_{OH}/V_{OL} specification. This is a typical part at room temperature, there will be a spread of other devices which will have voltages much closer to this limit. Of course, if your application doesn't need such tightly controlled voltages, then you can source or sink more current from the GPIO than the selected drive strength setting, but experimentation will be required to determine if it indeed safe to do so in your application, as it will be outside the scope of this specification.

5.5.3.6. Pin IO Delays

These delays include PIO's input/output mapping logic, IO muxing, and the actual pad delays into a nominal load of 5 pF. Min/max is over the extremes of process variation, voltage (1.1 V \pm 10%) and temperature (-40 C to 125 C).

These delays assume an IOVDD of 1.8 V, with PADS_VSEL set. At IOVDD = 3.3 V, the delay is significantly lower, and the range is smaller.

The flops themselves have a typical setup time of 10.6 ps and hold time of 2.2 ps. The IO delays between flops and pads must be taken into account.

For minimum and maximum output delays, from CLK_SYS arriving at any flop in PIO to the data being valid at a particular GPIO pad see Table 629.

Table 629. Pin minimum and maximum delays from flop to pad, in nanoseconds.

Pad output	Min delay (ns)	Max delay (ns)
GPIO0	2.27	7.10
GPIO1	2.31	7.07

图171。GPIO输出的典型电流-电压曲线。

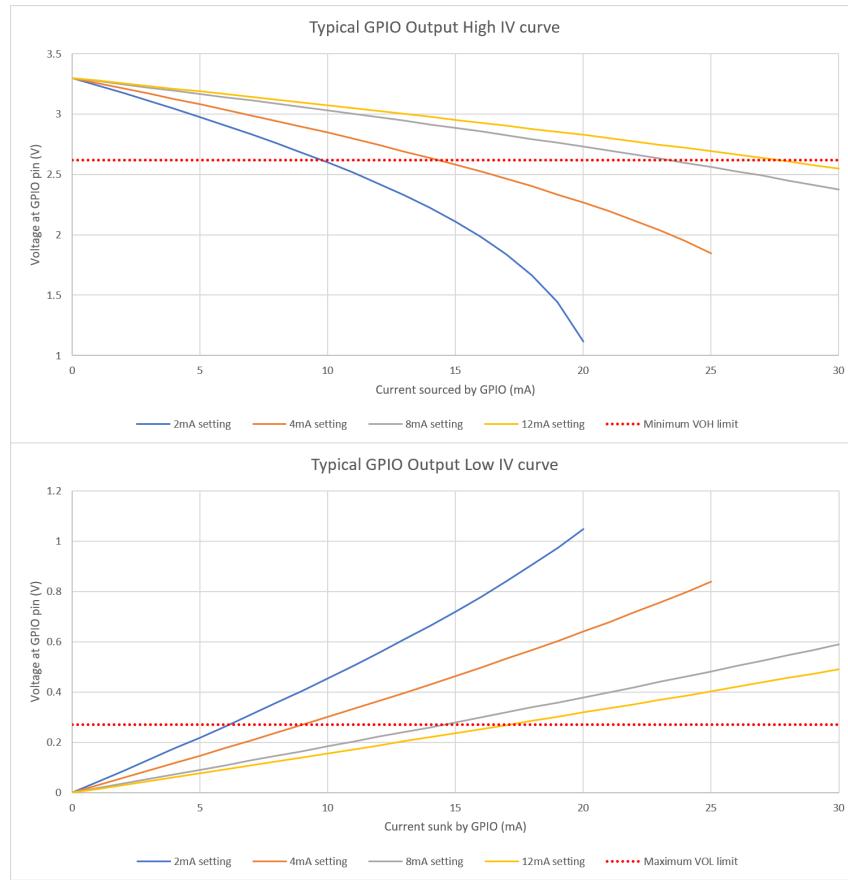


图171显示了随着引脚负载电流增加，对输出电压的影响。您可以清楚地观察不同驱动强度的影响；驱动强度越大，在给定电流下，输出电压越接近 IOVDD（或 0V）。最小 V_{OH} 和最大 V_{OL} 极限以红色标出。您可以看到，在每种驱动强度规定的电流下，电压均在许可极限之内，这意味着该特定器件能够驱动更大电流，且仍符合 V_{OH}/V_{OL} 规格。这是室温下的典型器件，其他器件的电压分布可能更接近该极限。当然，如果您的应用不需要如此严格控制的电压，则GPIO可以源出或吸入超过所选驱动强度设置的电流，但需要通过实验确定在您的应用中是否确实安全，因为这将超出本规范的范围。

5.5.3.6. 引脚IO延迟

这些延迟包括PIO的输入/输出映射逻辑、IO多路复用及实际垫片在标称负载5 pF下的延迟。最大/最小值涵盖工艺变化、电压（ $1.1\text{ V} \pm 10\%$ ）和温度（ -40°C 至 125°C ）的极限条件。

这些延迟假设IOVDD为1.8 V，且 [PADS_VSEL](#)已设置。IOVDD为3.3 V时，延迟显著降低且范围更小。

触发器本身的典型建立时间为10.6 ps，保持时间为2.2 ps。必须考虑触发器与垫片之间的IO延迟。

有关最小和最大输出延迟，从 [CLK_SYS](#)信号达到PIO中任一触发器至特定GPIO引脚数据有效的时间，详见表629。

表629。引脚从触发器到引脚的最小和最大延迟，单位：纳秒。

引脚输出	最小延迟（纳秒）	最大延迟（纳秒）
GPIO0	2.27	7.10
GPIO1	2.31	7.07

Pad output	Min delay (ns)	Max delay (ns)
GPIO2	2.33	7.08
GPIO3	2.24	7.00
GPIO4	2.30	7.07
GPIO5	2.34	7.10
GPIO6	2.32	7.10
GPIO7	2.39	7.09
GPIO8	2.34	7.09
GPIO9	2.38	7.08
GPIO10	2.33	7.07
GPIO11	2.36	7.08
GPIO12	2.35	7.04
GPIO13	2.31	7.08
GPIO14	2.38	7.06
GPIO15	2.33	7.05
GPIO16	2.34	7.09
GPIO17	2.37	7.09
GPIO18	2.37	7.04
GPIO19	2.27	7.10
GPIO20	2.38	7.09
GPIO21	2.05	7.10
GPIO22	2.34	7.07
GPIO23	2.16	7.05
GPIO24	2.12	7.06
GPIO25	2.26	7.10
GPIO26	2.32	7.09
GPIO27	2.26	7.08
GPIO28	2.34	7.09
GPIO29	2.30	7.07

For minimum and maximum input delays, from pad input to the input synchroniser, see [Table 630](#).

Table 630. Pin minimum and maximum delays from pad input to input synchroniser, in nanoseconds.

Pad output	Min delay (ns)	Max delay (ns)
GPIO1	1.89	5.22
GPIO2	1.84	5.25
GPIO3	1.83	5.24
GPIO4	1.90	5.17
GPIO5	1.90	5.14

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO2	2.33	7.08
GPIO3	2.24	7.00
GPIO4	2.30	7.07
GPIO5	2.34	7.10
GPIO6	2.32	7.10
GPIO7	2.39	7.09
GPIO8	2.34	7.09
GPIO9	2.38	7.08
GPIO10	2.33	7.07
GPIO11	2.36	7.08
GPIO12	2.35	7.04
GPIO13	2.31	7.08
GPIO14	2.38	7.06
GPIO15	2.33	7.05
GPIO16	2.34	7.09
GPIO17	2.37	7.09
GPIO18	2.37	7.04
GPIO19	2.27	7.10
GPIO20	2.38	7.09
GPIO21	2.05	7.10
GPIO22	2.34	7.07
GPIO23	2.16	7.05
GPIO24	2.12	7.06
GPIO25	2.26	7.10
GPIO26	2.32	7.09
GPIO27	2.26	7.08
GPIO28	2.34	7.09
GPIO29	2.30	7.07

有关最小和最大输入延迟，从引脚输入到输入同步器，详见表630。

表630。引脚
从引脚输入
到输入同步器的最
小和最大延迟
，单位：纳秒
。

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO1	1.89	5.22
GPIO2	1.84	5.25
GPIO3	1.83	5.24
GPIO4	1.90	5.17
GPIO5	1.90	5.14

Pad output	Min delay (ns)	Max delay (ns)
GPIO6	1.91	5.19
GPIO7	1.91	5.14
GPIO8	1.95	5.14
GPIO9	1.96	5.12
GPIO10	1.95	5.11
GPIO11	1.92	5.16
GPIO12	1.92	5.15
GPIO13	1.94	5.16
GPIO14	1.90	5.18
GPIO15	1.92	5.15
GPIO16	1.95	5.13
GPIO17	1.95	5.12
GPIO18	1.95	5.10
GPIO19	1.95	5.12
GPIO21	2.07	4.98
GPIO23	1.98	5.06
GPIO24	1.97	5.07
GPIO25	1.97	5.08
GPIO26	1.96	5.12
GPIO27	1.94	5.13
GPIO28	1.95	5.13
GPIO29	1.99	5.10

For minimum and maximum input delays over all corners, from pad input to state machine IN data flops (synchronisers bypassed) see [Table 631](#).

Table 631. Pin minimum and maximum delays from pad input to state machine IN data flops (synchronisers bypassed), in nanoseconds.

Pad input	Min delay (ns)	Max delay (ns)
GPIO1	2.22	5.45
GPIO2	2.25	5.49
GPIO3	2.23	5.18
GPIO4	2.24	5.41
GPIO5	2.30	5.65
GPIO6	2.25	5.48
GPIO7	2.26	5.50
GPIO8	2.30	5.51
GPIO9	2.25	5.68
GPIO10	2.34	5.71
GPIO11	2.28	5.47

引脚输出	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO6	1.91	5.19
GPIO7	1.91	5.14
GPIO8	1.95	5.14
GPIO9	1.96	5.12
GPIO10	1.95	5.11
GPIO11	1.92	5.16
GPIO12	1.92	5.15
GPIO13	1.94	5.16
GPIO14	1.90	5.18
GPIO15	1.92	5.15
GPIO16	1.95	5.13
GPIO17	1.95	5.12
GPIO18	1.95	5.10
GPIO19	1.95	5.12
GPIO21	2.07	4.98
GPIO23	1.98	5.06
GPIO24	1.97	5.07
GPIO25	1.97	5.08
GPIO26	1.96	5.12
GPIO27	1.94	5.13
GPIO28	1.95	5.13
GPIO29	1.99	5.10

有关在所有工况下，从引脚输入到状态机IN数据触发器（绕过同步器）的最小和最大输入延迟，详见表631。

表631。引脚
从引脚输入
到状态机IN数据触
发器（绕过同
步器）的最小和最
大延迟，单
位：纳秒。

引脚输入	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO1	2.22	5.45
GPIO2	2.25	5.49
GPIO3	2.23	5.18
GPIO4	2.24	5.41
GPIO5	2.30	5.65
GPIO6	2.25	5.48
GPIO7	2.26	5.50
GPIO8	2.30	5.51
GPIO9	2.25	5.68
GPIO10	2.34	5.71
GPIO11	2.28	5.47

Pad input	Min delay (ns)	Max delay (ns)
GPIO12	2.29	5.40
GPIO13	2.25	5.47
GPIO14	2.24	5.41
GPIO15	2.23	5.47
GPIO16	2.30	5.42
GPIO17	2.28	5.44
GPIO18	2.28	5.34
GPIO19	2.30	5.50
GPIO21	2.16	5.79
GPIO23	2.33	5.53
GPIO24	2.28	5.60
GPIO25	2.29	5.53
GPIO26	2.28	5.38
GPIO27	2.27	5.39
GPIO28	2.24	5.28
GPIO29	2.33	5.47

5.5.3.6.1. Effects of IOVDD

All of the IO delays given above assume IOVDD = 1.8V. Increasing IOVDD to 3.3V reduces the pad delays quite significantly, and the pad delay is a large fraction of the delays reported above. See [Table 632](#) for a summary of best and worst pad delays at 1.8V and 3.3V.

Table 632. Best and worst pad delays at 1.8V and 3.3V.

Path type	IOVDD	Min delay (ns)	Max delay (ns)
Output	1.8V	1.54	3.65
Output	3.3V	1.11	2.14
Input	1.8V	0.63	1.06
Input	3.3V	0.47	0.76

Changing IOVDD does not affect any logic in the core domain, so these differences can be added to the IO delay tables above to estimate the IO delay ranges at IOVDD = 3.3V (see [Table 633](#)).

Table 633. IO delay ranges at IOVDD = 3.3V.

Path group	IOVDD	Min delay (ns)	Max delay (ns)
Output	1.8V	2.12	7.10
Output	3.3V	1.69	5.59
Input to sync	1.8V	1.83	5.25
Input to sync	3.3V	1.67	4.95
Input to SM	1.8V	2.16	5.79
Input to SM	3.3V	2.00	5.49

引脚输入	最小延迟 (纳秒)	最大延迟 (纳秒)
GPIO12	2.29	5.40
GPIO13	2.25	5.47
GPIO14	2.24	5.41
GPIO15	2.23	5.47
GPIO16	2.30	5.42
GPIO17	2.28	5.44
GPIO18	2.28	5.34
GPIO19	2.30	5.50
GPIO21	2.16	5.79
GPIO23	2.33	5.53
GPIO24	2.28	5.60
GPIO25	2.29	5.53
GPIO26	2.28	5.38
GPIO27	2.27	5.39
GPIO28	2.24	5.28
GPIO29	2.33	5.47

5.5.3.6.1. IOVDD的影响

上述所有IO延迟均基于IOVDD = 1.8V的假设。提高IOVDD至3.3V将显著降低引脚延迟，而引脚延迟在上述延迟中占有较大比重。请参见表 632，了解在 1.8V 和 3.3V 下最佳与最差引脚延迟的汇总。

表 632。1.8V 和 3.3V 下的最佳与最差引脚延迟。

路径类型	IOVDD	最小延迟 (纳秒)	最大延迟 (纳秒)
输出	1.8V	1.54	3.65
输出	3.3V	1.11	2.14
输入	1.8V	0.63	1.06
输入	3.3V	0.47	0.76

更改 IOVDD 不影响核心域内的任何逻辑，因此可将这些差异加至上述 IO 延迟表中，以估算 IOVDD = 3.3V 时的 IO 延迟范围（详见表 633）。

表 633。IO 延迟
IO 延迟范围，IOVDD = 3.3V 时。

路径组	IOVDD	最小延迟 (纳秒)	最大延迟 (纳秒)
输出	1.8V	2.12	7.10
输出	3.3V	1.69	5.59
输入至同步	1.8V	1.83	5.25
输入至同步	3.3V	1.67	4.95
输入至 SM	1.8V	2.16	5.79
输入至 SM	3.3V	2.00	5.49

5.6. Power Supplies

Table 634. Power Supply Specifications

Power Supply	Supplies	Min	Typ	Max	Units
IOVDD ^a	Digital IO	1.62	1.8 / 3.3	3.63	V
DVDD ^b	Digital core	1.05	1.1	1.16	V
VREG_VIN	Voltage regulator	1.62	1.8 / 3.3	3.63	V
USB_VDD	USB PHY	3.135	3.3	3.63	V
ADC_AVDD ^c	ADC	1.62	3.3	3.63	V

^a If IOVDD <2.5V, GPIO VOLTAGE_SELECT registers should be adjusted accordingly. See [Section 2.9](#) for details.

^b Short term transients should be within +/-100mV. If using 200MHz for `clk_sys` as described in [Section 2.15.3](#), set DVDD to 1.15V.

^c ADC performance will be compromised at voltages below 2.97V

5.7. Power Consumption

5.7.1. Peripheral power consumption

Baseline readings are taken with only clock sources and essential peripherals (`BUSCTRL`, `BUSFAB`, `VREG`, Resets, ROM, SRAMs) active in the `WAKE_EN0`/`WAKE_EN1` registers. Clocks are set to default clock settings. Each peripheral is activated in turn by enabling all clock sources for the peripheral in the `WAKE_EN0`/`WAKE_EN1` registers. Current consumption is the increase in current when the peripheral clocks are enabled.

Table 635. Baseline power consumption

Peripheral	Typical DVDD Current Consumption ($\mu\text{A}/\text{MHz}$)
DMA	2.6
I2C0	3.9
I2C1	3.8
IO + Pads	23.6
PIO0	12.3
PIO1	12.5
PWM	5.0
RTC	1.1
SIO	1.9
SPI0	1.7
SPI1	1.8
Timer	1.2
UART0	3.5
UART1	3.7
Watchdog	1.0
XIP	37.6

5.6. 电源供应

表 634。电源规格

电源	电源供应	最小值	典型值	最大值	单位
IOVDD ^a	数字输入输出	1.62	1.8 / 3.3	3.63	V
DVDD ^b	数字核心	1.05	1.1	1.16	V
VREG_VIN	电压调节器	1.62	1.8 / 3.3	3.63	V
USB_VDD	USB PHY	3.135	3.3	3.63	V
ADC_AVDD ^c	ADC	1.62	3.3	3.63	V

^a 如果 IOVDD < 2.5V，GPIO VOLTAGE_SELECT 寄存器应相应调整。详见第 2.9 节。

^b 短期瞬态应控制在 +/-100mV 以内。如第 2.15.3 节所述使用 200MHz 作为 clk_sys，DVDD 应设为 1.15V。

^c ADC 性能在电压低于 2.97V 时将受损。

5.7. 功耗

5.7.1. 外设功耗

基线读数在 WAKE_EN0/WAKE_EN1 寄存器仅激活时钟源及基本外设 (BUSCTRL、BUSFAB、VREG、复位、ROM、SRAM) 状态下采集。时钟设为默认时钟配置。对于各外设，通过在 WAKE_EN0/WAKE_EN1 寄存器中启用该外设所有时钟源，依次激活。当前电流消耗是启用外设时电流的增量。

表 635。基线功耗

外设	典型 DVDD 电流消耗 ($\mu\text{A}/\text{MHz}$)
DMA	2.6
I2C0	3.9
I2C1	3.8
IO + 垫片	23.6
PIO0	12.3
PIO1	12.5
PWM	5.0
RTC	1.1
SIO	1.9
SPI0	1.7
SPI1	1.8
定时器	1.2
UART0	3.5
UART1	3.7
看门狗	1.0
XIP	37.6

Because of fixed external reference clocks of 48 MHz, as well as the variable system clock input, ADC and USBCTRL power consumption does not vary linearly with system clock (as it does for other peripherals which only have system and/or peripheral clock inputs). Absolute DVDD current consumption of the ADC and USBCTRL blocks at standard clocks (system clock of 125 MHz) is given below:

Table 636. Baseline power consumption for ADC and USBCTRL

Peripheral	Typical DVDD Current Consumption ($\mu\text{A}/\text{MHz}$)
ADC	0.1
USBCTRL	1.3

5.7.2. Power consumption for typical user cases

The following data shows the current consumption of various power supplies on 3 each of typical (**tt**), fast (**ff**) and slow (**ss**) corner RP2040 devices, with four different software use-cases.

NOTE

For power consumption of the Raspberry Pi Pico, please see the [Raspberry Pi Pico Datasheet](#).

Firstly, 'Popcorn' (Media player demo) using the VGA, SD Card, and Audio board. This demo uses VGA video, I2S audio and 4-bit SD Card access, with a system clock frequency of 48MHz.

NOTE

For more details of the VGA board see the [Hardware design with RP2040](#) book.

Secondly, the BOOTSEL mode of RP2040. These measurements are made both with and without USB activity on the bus, using a Raspberry Pi 4 as a host.

The third use-case uses the `hello_dormant` binary which puts RP2040 into a low power state, **DORMANT** mode.

The final use-case uses the `hello_sleep` binary code which puts RP2040 into a low power state, **SLEEP** mode.

Table 637 has two columns per power supply, 'Typical Average Current' and 'Maximum Average Current'. The former is the current averaged over several seconds that you might expect a typical RP2040 to consume at room temperature and nominal voltage (e.g., DVDD=1.1V, IOVDD=3.3V, etc). The 'Maximum Average Current' is the maximum current consumption (again averaged over several seconds) you might expect to see on a worst-case RP2040 device, across the temperature extremes, and maximum voltage (e.g., DVDD=1.21V, etc).

NOTE

The 'Popcorn' consumption measurements depend on the video being displayed at the time. The 'Typical' values are obtained over several seconds of video, with varied colour and intensity. The 'Maximum' values are measured during periods of white video, when the required current is at its highest.

Table 637. Power Consumption

Software Use-case	Typical Average DVDD Current	Max. Average DVDD current	Typical Average IOVDD Current	Max. Average IOVDD current	Typical Average USB_VDD Current	Max. Average USB_VDD current	Units
Popcorn	10.9	16.6	24.8	35.5	-	-	mA
BOOTSEL mode - Active	9.4	14.7	1.2	4.3	1.4	2.0	mA
BOOTSEL mode - Idle	9.0	14.3	1.2	4.3	0.2	0.6	mA
Dormant	0.18	4.2	-	-	-	-	mA

由于固定的 48 MHz 外部参考时钟及可变系统时钟输入，ADC 和 USBCTRL 的功耗不会随着系统时钟线性变化（而其他仅具有系统和/或外设时钟输入的外设则会如此）。在标准时钟（系统时钟 125 MHz）下，ADC 和 USBCTRL 模块的绝对 D VDD 电流消耗如下：

表 636。ADC 和 USBCTRL 的基线功耗

外设	典型 DVDD 电流消耗 ($\mu\text{A}/\text{MHz}$)
ADC	0.1
USBCTRL	1.3

5.7.2. 典型用户案例功耗

以下数据展示了三种典型（ tt ）、快速（ ff ）和慢速（ ss ）RP2040 器件在四种不同软件使用场景下，各电源的电流消耗情况。

i 注意

有关 Raspberry Pi Pico 的功耗详情，请参见 [Raspberry Pi Pico 数据手册](#)。

首先，“Popcorn”（媒体播放器演示）使用了VGA、SD卡和音频扩展板。该演示采用VGA视频、I2S音频以及4位SD卡访问，系统时钟频率为48MHz。

i 注意

有关VGA扩展板的更多详情，请参阅《[RP2040硬件设计](#)》一书。

其次，是RP2040的BOOTSEL模式。这些测量分别在总线有无USB活动的情况下进行，使用Raspberry Pi 4作为主机。

第三个用例使用了`hello_dormant`二进制文件，该文件使RP2040进入低功耗状态，即休眠模式。

最后一个用例使用了`hello_sleep`二进制代码，使RP2040进入低功耗状态，即睡眠模式。

表637为每个电源供应列出了两栏：“典型平均电流”和“最大平均电流”。前者为在室温和标称电压条件下（例如DVDD=1.1V, IOVDD=3.3V等），典型RP2040在数秒内的平均电流。“最大平均电流”指在极端温度和最高电压条件下（例如 DVDD=1.21V 等），最坏情况的 RP2040 设备上可能出现的最大电流消耗（为数秒的平均值）。

i 注意

“爆米花”电流消耗测量取决于当前显示的视频内容。“典型”值为播放多种颜色和强度视频数秒后的平均电流。“最大”值为播放白色视频期间测得的电流，此时电流需求最高。

表637. 功耗

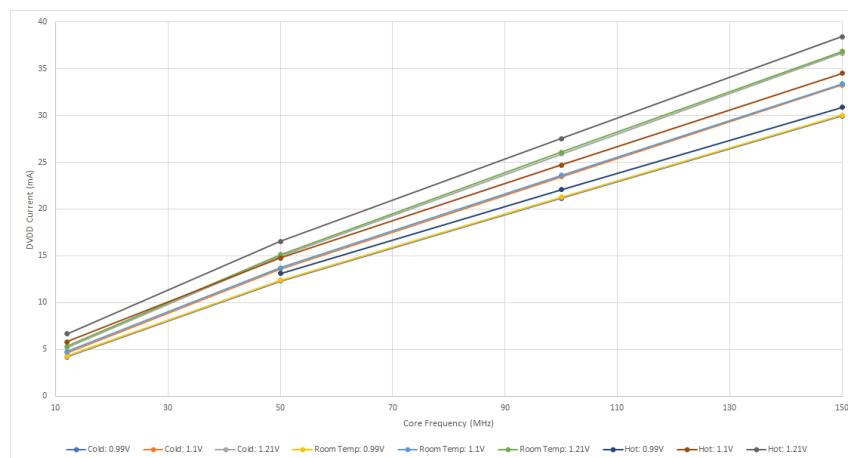
软件使用案例	典型平均 DVDD 电流	最大平均 DVDD 电流	典型平均 IOVDD 电流	最大平均 IOVDD 电流	典型平均 USB_VDD 电流	最大平均 USB_VDD 电流	单位
爆米花	10.9	16.6	24.8	35.5	-	-	mA
BOOTSEL 模式 - 活跃	9.4	14.7	1.2	4.3	1.4	2.0	mA
BOOTSEL 模式 - 空闲	9.0	14.3	1.2	4.3	0.2	0.6	mA
休眠	0.18	4.2	-	-	-	-	mA

Software Use-case	Typical Average DVDD Current	Max. Average DVDD current	Typical Average IOVDD Current	Max. Average IOVDD current	Typical Average USB_VDD Current	Max. Average USB_VDD current	Units
Sleep	0.39	4.5	-	-	-	-	mA

5.7.2.1. Power Consumption versus frequency

To give an indication of the relationship between the core frequency that RP2040 is operating at, and the current consumed by the DVDD supply, [Figure 172](#) shows the measured results of a typical RP2040 device, continuously running FFT calculations on both cores, at various core clock frequencies. [Figure 172](#) also shows the effects of case temperature, and DVDD voltage upon the current consumption.

Figure 172. DVDD Current vs Core Frequency of a typical RP2040 device, whilst running FFT calculations

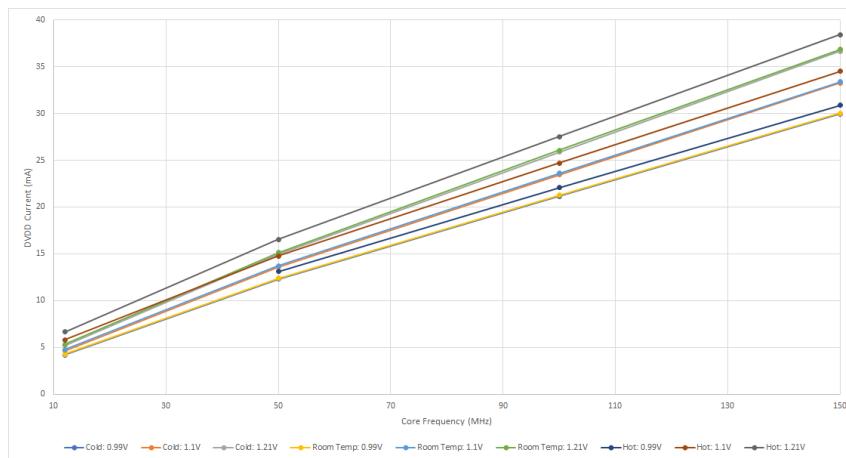


软件使用案例	典型 平均 DVDD 电流	最大平均 DVDD 电流	典型 平均 IOVDD 电流	最大平均 IOVDD 电流	典型 平均 USB_VDD 电流	最大平均 USB_VDD 电流	单位
睡眠	0.39	4.5	-	-	-	-	mA

5.7.2.1. 功耗与频率的关系

为表明RP2040核心运行频率与DVDD电源电流消耗之间的关系，图172展示了典型RP2040器件在各核心时钟频率下，双核持续执行FFT计算的测量结果。图172还展示了封装温度及DVDD电压对电流消耗的影响。

图172。典型RP2040器件运行FFT计算时，DVDD电流与核心频率的关系



Appendix A: Register Field Types

Standard types

RW:

- Read/Write
- Read operation returns the register value
- Write operation updates the register value

RO:

- Read-only
- Read operation returns the register value
- Write operations are ignored

WO:

- Write-only
- Read operation returns 0
- Write operation updates the register value

Clear types

SC

- Self-Clearing
- Writing a 1 to a bit in an SC field will trigger an event, once the event is triggered the bit clears automatically
- Writing a 0 to a bit in an SC field does nothing

WC

- Write-Clear
- Writing a 1 to a bit in a WC field will write that bit to 0
- Writing a 0 to a bit in a WC field does nothing
- Read operation returns the register value

附录A：寄存器字段类型

标准类型

RW:

- 读/写
- 读操作返回寄存器的值
- 写操作更新寄存器的值

RO:

- 只读
- 读操作返回寄存器的值
- 写操作将被忽略

WO:

- 只写
- 读操作返回0
- 写操作更新寄存器的值

清除类型

SC

- 自清除
- 向SC字段中的位写入1将触发事件，事件触发后该位会自动清零
- 向SC字段中的位写入0不会执行任何操作

WC

- 写-清除
- 向WC字段中的位写入1会将该位写为0
- 向WC字段中的位写入0不产生任何效果
- 读操作返回寄存器的值

FIFO types

These fields are used for reading and writing data to and from FIFOs. Accompanying registers provide FIFO control and status. There is no fixed format for the control and status registers, as they are specific to each FIFO interface.

RWF

- Read/Write FIFO
- Reading this field returns data from a FIFO
 - When the read is complete, the data value is removed from the FIFO
 - If the FIFO is empty, a default value will be returned; the default value is specific to each FIFO interface
- Data written to this field is pushed to a FIFO, Behaviour when the FIFO is full is specific to each FIFO interface
- Read and write operations may access different FIFOs

RF

- Read FIFO
- Functions the same as RWF, but read-only

WF

- Write FIFO
- Functions the same as RWF, but write-only

FIFO 类型

这些字段用于从 FIFO 读写数据，配套寄存器提供 FIFO 的控制和状态信息。控制和状态寄存器无固定格式，因其针对各 FIFO 接口特定设计。

RWF

- 读/写 FIFO
- 读取该字段返回来自 FIFO 的数据
 - 读取完成后，数据值将从 FIFO 中移除
 - 如 FIFO 为空，将返回默认值；默认值因各 FIFO 接口而异
- 写入此字段的数据将被推入 FIFO，FIFO 满时的行为由各 FIFO 接口具体定义
- 读写操作可能访问不同的 FIFO

RF

- 读 FIFO
- 功能与 RWF 相同，但只读

WF

- 写 FIFO
- 功能与 RWF 相同，但只写

Appendix B: Errata

Hardware blocks are listed alphabetically. Errata are listed numerically under the relevant block.

Bootrom

RP2040-E9

Reference	RP2040-E9
Summary	ROM bootloader cannot boot directly into XIP cache-as-SRAM
Description	<p>The XIP cache can be used as an additional 16kB SRAM bank when XIP caching is disabled (Section 2.6.3.1). The UF2 bootloader supports RAM-only UF2 binaries, which it loads directly into memory, and enters via a watchdog reboot. A single UF2 binary can initialise both the XIP cache contents and main system memory, and the cache is disabled by the bootloader, so that cache contents be written.</p> <p>However, the watchdog reset re-enables the cache, so booting directly into the cache-as-SRAM alias causes an immediate bus fault. The cache contents are preserved, but can not be accessed immediately post-boot.</p>
Workaround	<p>Add code in main SRAM to re-disable XIP caching before accessing the cache-as-SRAM alias. When entering a RAM-only UF2 binary, the bootloader selects the lowest loaded address in either main SRAM or cache-as-SRAM as the entry point, preferring main SRAM if both are loaded.</p> <p>Additionally, if the <code>0x15...</code> segment is <i>written</i> immediately post-boot, a dummy read of the FLUSH register is required, so that no cache-as-SRAM writes take place during the tag memory flush triggered by the watchdog (see Section 2.6.3.2).</p>
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation

RP2040-E14

Reference	RP2040-E14
Summary	Sparse or mis-aligned flash-binary UF2 may not be written to flash correctly by the UF2 bootloader

附录 B：勘误

硬件模块按字母顺序排列。勘误按编号列于相关章节下。

Bootrom

RP2040-E9

参考	RP2040-E9
摘要	ROM引导程序无法直接引导进入作为SRAM的XIP缓存
描述	<p>当XIP缓存被禁用时（第2.6.3.1节），XIP缓存可用作额外的16kB SRAM块。UF2引导程序支持仅RAM的UF2二进制文件，直接加载到内存中，并通过看门狗复位进入。单个UF2二进制文件可以初始化XIP缓存内容与主系统内存，引导程序会禁用缓存，以便写入缓存内容。</p> <p>然而，看门狗复位会重新启用缓存，因此直接引导进入作为SRAM的缓存别名会导致立即发生总线错误。缓存内容得到保留，但启动后无法立即访问。</p>
解决方案	<p>在主SRAM中添加代码，在访问作为SRAM的缓存别名之前重新禁用XIP缓存。进入仅RAM的UF2二进制文件时，引导程序选择主SRAM或作为SRAM的缓存中较低的加载地址作为入口点，若两者均已加载，则优先选择主SRAM。</p> <p>此外，如果 <code>0x15...</code> 段在启动后立即写入，则需要对FLUSH寄存器执行虚读，以确保在看门狗触发的标签存储器刷新过程中不会发生缓存即SRAM的写入（参见第2.6.3.2节）。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档

RP2040-E14

参考	RP2040-E14
摘要	稀疏或未对齐的闪存二进制UF2可能无法被UF2引导程序正确写入闪存

Description	<p>A RP2040 UF2 file consists of 256-byte pages of data, each marked to be written at a certain address by the UF2 bootloader. A flash-binary UF2 is one of these for which every 256-byte page is marked to be written at a 256-byte-aligned address in flash.</p> <p>When writing flash, an entire 4kB flash sector must be erased at a time before any pages within that sector can be (re-)written. The UF2 bootloader does not require the flash-binary UF2 to include data for all pages within a sector. In that case the whole sector will first be erased, any present pages will be written, and the rest of the 4kB sector will be left undefined.</p> <p>This mechanism works as expected when the <i>partially-filled</i> sector is at the end of the binary, which is of course commonplace, as a binary does not need to be a multiple of 4kB long.</p> <p>If however, the <i>partially-filled</i> sector occurs at the start of the binary (i.e. the binary is not aligned on a 4kB page) or if a <i>partially-filled</i> sector appears in the middle of the binary (i.e. the binary is sparse/non-contiguous), then the UF2 file may be written incorrectly.</p> <p>Note that the vast majority of UF2s generated by the SDK are indeed aligned on a 4kB boundary and contiguous, however it <i>is</i> possible for the SDK to produce a misaligned or non-contiguous binary by modifying the linker scripts, or putting extreme alignment requirements on static data. It is also possible that other languages or tools might produce binaries that are not 4kB-aligned or contiguous.</p>
Workaround	<p>The workaround is to include data for <i>all</i> the pages in any 4kB sector (other than the last) that contains data for <i>some</i> pages.</p> <p>This is handled for you automatically by the elf2uf2 tool in the SDK version 1.3.1 onwards, which explicitly adds zero-filled pages to the appropriate <i>partially-filled</i> sectors.</p>
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation / Software

Clocks

RP2040-E7

Reference	RP2040-E7
Summary	ROSC and XOSC <code>COUNT</code> registers are unreliable
Description	The ROSC and XOSC <code>COUNT</code> registers are intended to be used in the configuration of components like PHYs and PLLs where microsecond scale delays are required and NOP loops are inadequate because the <code>clk_sys</code> frequency is variable. However due to a synchronisation issue the ROSC: <code>COUNT</code> and XOSC: <code>COUNT</code> registers are unreliable.
Workaround	Do not use ROSC: <code>COUNT</code> or XOSC: <code>COUNT</code>
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Not fixed, do not use. These registers are not used by the C SDK.

RP2040-E10

Reference	RP2040-E10
------------------	------------

描述	<p>RP2040的UF2文件由256字节的数据页组成，每页均标记为由UF2引导程序写入指定地址。闪存二进制UF2是指其每个256字节的数据页均被标记为写入闪存中256字节对齐地址的UF2文件。</p> <p>写入闪存时，必须先擦除整个4kB闪存扇区，随后方可对该扇区内的任意数据页进行（重新）写入。UF2引导加载程序不要求闪存二进制UF2必须包含某个扇区内所有页面的数据。在这种情况下，整个扇区将首先被擦除，随后写入存在的页面，4kB扇区的其余部分将保持未定义状态。</p> <p>当部分填充的扇区位于二进制文件末尾时，此机制能够如预期般正常工作；这非常常见，因为二进制文件不必是4kB的整数倍长度。</p> <p>然而，若部分填充的扇区出现在二进制文件开头（即二进制文件未按4kB页面对齐），或部分填充的扇区位于二进制文件中间（即二进制文件稀疏或不连续），则该UF2文件可能会被错误写入。</p> <p>请注意，SDK所生成的绝大多数UF2文件确实按4kB边界对齐且连续，然而通过修改链接脚本或对静态数据施加极端对齐要求，SDK亦可能产生非对齐或非连续的二进制文件。其他语言或工具生成的二进制文件也可能不满足4kB对齐或连续存放的要求。</p>
解决方案	<p>解决方案是在包含某些页面数据的任何4kB扇区（除最后一个外）中包含所有页面的数据。</p> <p>从SDK版本1.3.1起，elf2uf2工具会自动处理此问题，明确向适当的部分填充扇区添加填充零页。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档 / 软件

时钟

RP2040-E7

参考	RP2040-E7
摘要	ROSC和XOSC COUNT寄存器不可靠
描述	ROSC和XOSC COUNT寄存器设计用于配置PHY及PLL等组件，适用于需要微秒级延迟且clk_sys频率可变时，NOP循环不足以满足的情况。但由于同步问题，ROSC:COUNT和XOSC:COUNT寄存器不可靠。
解决方案	请勿使用ROSC:COUNT或XOSC:COUNT寄存器
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复，禁止使用。C SDK未使用这些寄存器。

RP2040-E10

参考	RP2040-E10
----	------------

Summary	BADWRITE field in ROSC STATUS register is unreliable
Description	The BADWRITE field in the ROSC STATUS register was intended to report when invalid values had been written to other ROSC registers. However due to internal bugs the ROSC: STATUS . BADWRITE field is unreliable.
Workaround	Do not use ROSC: STATUS . BADWRITE field
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Not fixed, do not use. This field is not used by the C SDK.

DMA

RP2040-E12

Reference	RP2040-E12
Summary	Reading DMA WRITE_ADDR and READ_ADDR registers when an address-wrapping or non-incrementing transfer sequence is in progress gives wrong values
Description	<p>The DMA's internal WRITE_ADDR and READ_ADDR registers are incremented every time the DMA issues a new address to its bus pipeline. If the processor reads these registers whilst a sequence of transfers is in progress, the value reported by the DMA is adjusted downward by the number of in-flight transfers (i.e. issued to the bus pipeline and not yet completed) times the individual transfer size in bytes.</p> <p>This logic was added to ensure that reading READ_ADDR and WRITE_ADDR reflects addresses where the read/write has <i>completed</i>, not merely where the address has been issued. This logic does not take into account that READ_ADDR and WRITE_ADDR do not increment linearly for some transfer modes, specifically, when <code>CTRL.INCR_WRITE == 0</code>, <code>CTRL.INCR_READ == 0</code> or <code>CTRL.RING_SIZE != 0</code>.</p>
Workaround	<p>Instead of checking READ_ADDR or WRITE_ADDR to monitor the progress of a transfer sequence, check TRANS_COUNT.</p> <p>TRANS_COUNT has similar in-flight adjustment logic, but is not affected by this erratum because it always decrements linearly. The correct values of READ_ADDR and WRITE_ADDR can be calculated based on their initial values and TRANS_COUNT.</p>
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation

RP2040-E13

Reference	RP2040-E13
Summary	After aborting a channel, the ABORT status clears prematurely, and an interrupt may be asserted
Description	<p>The DMA ABORT register is used to cancel an ongoing sequence of transfers, for example when a channel is stuck on an inactive peripheral DREQ. If, at the point the abort is triggered, the channel currently has any transfers in flight (i.e. the read cycle of the transfer has taken place, but the write cycle has not), the ABORT bit does not wait for these in-flight transfers to complete before clearing.</p> <p>When the in-flight transfers complete, because the ABORT bit was prematurely cleared, the DMA treats this as a normal completion. This sets the channel's interrupt status flag, assuming <code>CTRL.IRQ_QUIET</code> has not been set.</p>

摘要	ROSC STATUS寄存器中的BADWRITE字段不可靠
描述	ROSCSTATUS寄存器中的BADWRITE字段意在报告向其他ROSC寄存器写入无效值的情况。然而，由于内部缺陷，ROSC:STATUS.BADWRITE字段不可靠。
解决方案	请勿使用ROSC:STATUS.BADWRITE字段。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复，禁止使用。C SDK不使用该字段。

DMA

RP2040-E12

参考	RP2040-E12
摘要	在地址环绕或非递增传输序列进行时，读取DMA的WRITE_ADDR和READ_ADDR寄存器会返回错误值。
描述	<p>DMA的内部WRITE_ADDR和READ_ADDR寄存器会在每次DMA向其总线流水线发出新地址时递增。若处理器在传输序列进行中读取这些寄存器，DMA返回的值将被调整，按进行中的传输数量（即已发送至总线流水线但尚未完成的传输）乘以单次传输字节数向下修正。</p> <p>该逻辑旨在确保读取READ_ADDR和WRITE_ADDR反映读/写操作已完成的位置，而非仅仅是地址已发出的位置。该逻辑未考虑到某些传输模式——尤其是当CTRL.INCR_WRITE == 0、CTRL.INCR_READ == 0或CTRL.RING_SIZE != 0时，READ_ADDR和WRITE_ADDR不会线性递增。</p>
解决方案	<p>与其检查READ_ADDR或WRITE_ADDR来监控传输序列的进度，不如检查TRANS_COUNT。</p> <p>TRANS_COUNT具有类似的传输中调整逻辑，但不受此缺陷影响，因为它始终线性递减。可根据READ_ADDR和WRITE_ADDR的初始值及TRANS_COUNT计算它们的正确值。</p>
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档

RP2040-E13

参考	RP2040-E13
摘要	在中止通道后，ABORT状态会被过早清除，且可能触发中断。
描述	<p>DMA的ABORT寄存器用于取消正在进行的传输序列，例如当通道卡在非活动的外设DREQ请求时。如果在触发中止时通道存在任何进行中的传输（即传输的读取周期已完成但写入周期未完成），则ABORT位不会等待这些传输完成即被清除。</p> <p>由于ABORT位被过早清除，当进行中的传输完成时，DMA将其视为正常完成。此操作设置通道的中断状态标志，前提是未设置CTRL.IRQ_QUIET。</p>

Workaround	Before aborting a channel, clear its interrupt enable. After aborting a channel, poll the <code>CTRL.BUSY</code> bit to wait for completion (not the <code>ABORT</code> bit), clear the spurious IRQ, and restore the interrupt enable.
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Software

GPIO / ADC

RP2040-E6

Reference	RP2040-E6
Summary	GPIO digital inputs not disabled for ADC pins by default
Description	GPIO26-29 are shared with ADC inputs AIN0-3. The GPIO digital input is enabled after RUN is released. If the pins are connected to an analogue signal to measure, there could be unexpected signal levels on these pads. This is unlikely to cause a problem as the digital inputs have hysteresis enabled by default.
Workaround	If analogue inputs are used, the digital input should be disabled as early as possible after startup. This is done in the RP2040B2 bootrom and early on in SDK platform setup code on RP2040B0 and RP2040B1. If user wishes to use digital inputs, they must be enabled.
Affects	RP2040B0, RP2040B1
Fixed by	RP2040B2 bootrom. Fixed on RP2040B0 and RP2040B1 in SDK. Custom user code should disable these inputs early on.

RP2040-E11

Reference	RP2040-E11
Summary	DNL error peaks in ADC
Description	The RP2040 ADC has a DNL that is mostly flat, and below 1 LSB. However at four values – 512, 1,536, 2,560, and 3,584 – the ADC’s DNL error peaks above this value. The ENOB for the ADC has been reduced from 9-bits (simulated) to 8.7-bits (measured), see Section 4.9.3 . The DNL errors will somewhat limit the performance of the ADC dependent on use case.
Workaround	None
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Not fixed.

USB

RP2040-E2

Reference	RP2040-E2
Summary	USB device endpoint abort is not cleared.

解决方案	在中止通道之前，应清除其中断使能。中止通道后，应轮询 CTRL.BUSY 位以等待操作完成（非 ABORT 位），清除虚假中断，并恢复中断使能。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

GPIO / ADC

RP2040-E6

参考	RP2040-E6
摘要	默认情况下，ADC引脚的GPIO数字输入未被禁用。
描述	GPIO26-29与ADC输入AIN0-3共用。RUN释放后，GPIO数字输入被启用。若将这些引脚接入模拟信号进行测量，可能会在这些引脚上产生意外的信号电平。通常这不会引起问题，因为数字输入默认启用迟滞效应。
解决方案	如使用模拟输入，应在启动后尽早禁用数字输入。此操作由RP2040B2的引导ROM以及RP2040B0和RP2040B1 SDK平台初始化代码的早期阶段完成。用户若需使用数字输入，必须手动启用。
影响	RP2040B0, RP2040B1
修复于	RP2040B2 启动只读存储器。该问题已在 SDK 中的 RP2040B0 和 RP2040B1 上修复。自定义用户代码应尽早禁用相关输入。

RP2040-E11

参考	RP2040-E11
摘要	ADC 中的 DNL 误差峰值
描述	RP2040 ADC 的 DNL 大致平坦且低于 1 LSB，但在四个数值——512、1536、2560 和 3584——处，其 DNL 误差峰值超过该值。ADC 的有效位数 (ENOB) 已由模拟的 9 位降低至实测的 8.7 位，详见第 4.9.3 节。DNL 误差将在一定程度上依据应用场景限制 ADC 性能。
解决方案	无
影响	RP2040B0, RP2040B1, RP2040B2
修复于	未修复。

USB

RP2040-E2

参考	RP2040-E2
摘要	USB 设备端点中止状态未被清除。

Description	The USB device controller (Section 4.1) has the ability to abort any pending transactions on an endpoint by setting that endpoint's bit in the EP_ABORT register. Due to a logic error, the USB device controller will reply with NAKs forever on all endpoints if a transaction is initiated for any endpoint with the EP_ABORT bit set.
Workaround	Do not use the EP_ABORT bits.
Affects	RP2040B0, RP2040B1
Fixed by	RP2040B2

RP2040-E3

Reference	RP2040-E3
Summary	USB host: interrupt endpoint buffer done flag can be set with incorrect buffer select.
Description	The USB host has two types of transactions: normal software initiated transfer, and interrupt transfers, where the host polls an interrupt endpoint after a specific amount of time. For example, polling a mouse every 1ms to check for movement. Interrupt transfer are single buffered, but the controller doesn't reset the buffer selector to zero. This means that if a software initiated transfer happened then the interrupt transfer can potentially raise the buffer done flag with BUF1 selected instead of BUF0 . The fix is to ignore the BUFF_CPU_SHOULD_HANDLE register for interrupt endpoints.
Workaround	
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Software

RP2040-E4

Reference	RP2040-E4
Summary	USB host writes to upper half of buffer status in single buffered mode.
Description	The USB host maintains a buffer selector which switches between BUF0 and BUF1 . This should only be toggled in double buffered mode but is toggled in single buffered mode too. For a transaction lasting multiple packets (i.e. length more than 8 bytes in low speed mode, and length more than 64 bytes in full speed mode), the buffer status can be written back to the BUF1 half of the status register when the buffer select is incorrectly set to BUF1 . Note this does not affect reading new buffer information from the buffer control register, as the controller ignores the buffer selector in single buffered mode when reading the buffer control register.
Workaround	Shift endpoint control register to the right by 16 bits if the buffer selector is BUF1 . You can use BUFF_CPU_SHOULD_HANDLE find the value of the buffer selector when the buffer was marked as done.
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Software

RP2040-E5

Reference	RP2040-E5
Summary	USB device fails to exit RESET state on busy USB bus.

描述	USB设备控制器（第4.1节）允许通过设置EP_ABORT寄存器中相应端点的位来中止该端点上任何待处理的事务。由于逻辑错误，如果对任何设置了EP_ABORT位的端点发起传输，USB设备控制器将不断对所有端点返回NAK。
解决方案	请勿使用EP_ABORT位。
影响	RP2040B0, RP2040B1
修复于	RP2040B2

RP2040-E3

参考	RP2040-E3
摘要	USB主机：中断端点的缓冲区完成标志可能在错误的缓冲区选择情况下被设置。
描述	USB主机有两种传输类型：正常的软件启动传输和中断传输，其中主机会在特定时间间隔后轮询中断端点。例如，每1毫秒轮询一次鼠标以检测其移动。中断传输为单缓冲模式，但控制器不会将缓冲区选择器重置为零。这意味着，如果发生软件启动传输，中断传输可能在选择了BUF1而非BUF0时触发缓冲完成标志。解决方案是忽略中断端点的BUFF_CPU_SHOULD_HANDLE寄存器。
解决方案	
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

RP2040-E4

参考	RP2040-E4
摘要	USB主机在单缓冲模式下写入缓冲状态的高半部分。
描述	USB主机维护一个缓冲区选择器，用于在BUF0和BUF1之间切换。此项设置应仅在双缓冲模式下切换，但在单缓冲模式下亦被切换。对于持续多个数据包的传输（即低速模式下长度超过8字节，全速模式下长度超过64字节），当缓冲区选择错误地设置为BUF1时，缓冲区状态可写回状态寄存器的BUF1半区。请注意，这不会影响从缓冲控制寄存器读取新的缓冲区信息，因为控制器在单缓冲模式下读取该寄存器时会忽略缓冲区选择器。
解决方案	如果缓冲区选择器为BUF1，则应将端点控制寄存器右移16位。您可以使用BUFF_CPU_SHOULD_HANDLE查找缓冲区被标记完成时的缓冲区选择器值。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	软件

RP2040-E5

参考	RP2040-E5
摘要	USB设备无法在繁忙的USB总线上退出RESET状态。

Description	<p>The USB bus RESET state is triggered by the host sending SE0 for 10ms to the device. The USB device controller requires 800µs of idle (J-state) after a bus reset before moving to the CONNECTED state. Without this idle time, the USB device does not connect and will not receive any packets from the host, and so does not enumerate.</p> <p>A device reset happens just after the device is plugged in. Although a host will wait before talking to a newly-reset device, other devices attached to the same USB hub may also be communicating with the host.</p> <p>USB 2.0 and USB 3.0 hubs have one or more transaction translators, which facilitate low speed and full speed transactions on a higher speed bus. It depends on the hub design, but a transaction translator is usually shared between a few ports.</p> <p>As the RP2040 USB device is full speed, its traffic when connected to a hub will come via a transaction translator. This means that if you have another device plugged in next to an RP2040, the RP2040 is likely to see some messages from the host addressed to the other device. If the device is not very active, for example, a mouse that is polled every 8ms, this is not a problem. However some devices, such as a USB serial port, are polled every 30-50µs. In this case the bus is very active, and will cause the RP2040 to never exit RESET state and not connect.</p> <p>There is a hardware fix in RP2040B2 which avoids the need for 800µs of IDLE time after RESET state.</p> <p>There is a software workaround for this issue (see workaround section). A user can also work around this by closing the USB serial port or any other offending devices while connecting their RP2040 and then re-opening their USB serial port.</p> <p>On a larger hub, the problem may be fixed by moving the RP2040 far away (onto a different transaction translator) from the offending device. For example, connecting the RP2040 to port 1 of a 7 port hub, and connecting the USB serial console to port 7, may solve the issue. Connecting the RP2040 to a separate USB hub to any busy devices will also fix the problem.</p>
Workaround	<p>Use software to force USB device controller to see idle USB bus for 800µs to move the device from the RESET state to the CONNECTED state. This fix uses internal debug logic that is connected to GPIO15 for a short amount of time (~800µs). This forces the controller to see DP as a logical 1 (and DM as logical 0) to make the USB Device controller believe there is a J-state on the USB bus. GPIO15 does not need to be tied in any particular way for this fix to work. Instead, we can force the input path in software using the Section 2.19 input override feature. See https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_fix/rp2040_usb_device_enumeration/rp2040_usb_device_enumeration.c.</p> <p>NOTE The workaround takes control of GPIO 15 during a device reset, so you need to be sure that you are not using GPIO 15 for anything else during a device reset before using the workaround. A device reset happens after the first connection, but may also happen at other times under the host's control.</p> <p>Using the workaround with TinyUSB and the SDK is easy, as the above source file is included by the library <code>pico_fix_rp2040_usb_device_enumeration</code> (which is automatically added as a dependency of TinyUSB in device mode). The fix itself is still off by default though, since the fix's use of GPIO 15 may conflict with the application's own use of GPIO 15. You can enable it by setting either <code>PICO_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> as part of your compiler definitions in your <code>CMakeLists.txt</code>, or <code>TUD_OPT_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> in your <code>tusb_config.h</code>.</p> <p>It is safe (and inexpensive) to enable the software workaround even when using versions of RP2040 which include the fix in hardware.</p>
Affects	RP2040B0, RP2040B1

描述	<p>USB总线的 RESET 状态由主机向设备发送持续10毫秒的 SE0信号触发。USB设备控制器在总线复位后需要800μs的空闲时间（J态），然后才能进入 CONNECTED状态。如果没有这段空闲时间，USB设备将无法连接，且不会接收主机发来的任何数据包，从而无法完成枚举。</p> <p>设备复位发生于设备插入之后。尽管主机会在与刚复位的设备通信之前进行等待，但接入同一USB集线器的其他设备可能已在与主机通信。</p> <p>USB 2.0和USB 3.0集线器配备一个或多个事务转换器，以便在高速总线上支持低速和全速事务。具体情况取决于集线器设计，但一个事务转换器通常由多个端口共享。</p> <p>由于RP2040的USB设备为全速设备，当其连接到集线器时，其通信流量将通过事务转换器传输。这意味着如果在RP2040旁边连接了另一设备，RP2040可能会接收到主机发给该设备的一些消息。如果设备活动不频繁，例如每8毫秒轮询一次的鼠标，则不会出现问题。然而某些设备，如USB串口，每30至50μs轮询一次。在这种情况下，总线活动非常频繁，会导致RP2040无法退出 RESET 状态并无法连接。</p> <p>RP2040B2中有硬件修正，避免了在 RESET 状态后需800μs空闲时间的情况。</p> <p>对此问题也有软件解决方案（详见解决方案部分）。用户还可通过在连接RP2040时关闭USB串口或其他有问题的设备，随后重新打开USB串口以规避该问题。</p> <p>在更大型的集线器中，将RP2040移至远离问题设备（即不同事务转换器）处，也可能解决该问题。例如，将RP2040连接至7端口集线器的端口1，将USB串口控制台连接至端口7，可能解决该问题。将RP2040连接至独立USB集线器上的任何繁忙设备，亦可解决该问题。</p>
解决方案	<p>通过软件强制USB设备控制器检测空闲USB总线800μs，将设备从 RESET 状态切换至 CONNECTED 状态。该修复利用连接至GPIO15的内部调试逻辑，作用时间为800μs。此举强制控制器将DP视为逻辑1（DM为逻辑0），使USB设备控制器判定USB总线上存在 J-state。此修复无须对GPIO15进行特定连接。我们亦可使用第2.19节中的输入覆盖功能，通过软件强制输入路径。详见 https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_fix/rp2040_usb_deviceEnumeration/rp2040_usb_device_enumeration.c。</p> <p>注意该方案在设备重置期间会控制GPIO 15，使用前请确保设备重置阶段未将GPIO 15用于其他用途。设备在首次连接后会发生复位，但主机控制下的其他时间也可能发生复位。</p> <p>使用TinyUSB和SDK的解决方案十分简便，因为上述源文件已被 <code>pico_fix_rp2040_usb_device_enumeration</code> 库包含（该库作为设备模式下TinyUSB的自动依赖项添加）。该修复默认未启用，因修复方案使用GPIO 15，可能与应用程序自身对GPIO 15的使用产生冲突。您可以通过设置以下任一选项来启用该修复</p> <p>在您的 <code>CMakeLists.txt</code> 中将 <code>PICO_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> 加入编译器定义，或 在您的 <code>tusb_config.h</code> 中将 <code>TUD_OPT_RP2040_USB_DEVICE_ENUMERATION_FIX=1</code> 设置。</p> <p>即便使用内含硬件修复的RP2040版本，启用软件解决方案仍然安全且代价低廉。</p>
影响	RP2040B0, RP2040B1

Fixed by	RP2040B2. Software workaround on RP2040B0, RP2040B1. The workaround isn't present in the USB mass storage code in the bootrom. The software workaround requires use of GPIO15 during USB bus reset.
-----------------	---

RP2040-E15

Reference	RP2040-E15
Summary	USB Device controller will hang if certain bus errors occur during an IN transfer.
Description	<p>The USB Device controller enters an unrecoverable state if the following critical sequence of events occurs:</p> <ul style="list-style-type: none"> • RP2040 is connected to a VL805 xHCI controller and is operating in full-speed mode • The integrated hub detects an impending line collision between downstream port Transaction Translator traffic and broadcast upstream traffic (Start-of-Frame token) • The integrated hub forces a bitstuffing error during the PID or CRC portions of the downstream in-progress packet or token. <p>This sequence is known to occur with the downstream-facing ports on a Raspberry Pi 4 or a Raspberry Pi 400 and Bulk IN endpoints with data buffer sizes of more than 50 bytes. In this case, the integrated USB2.0 hub incorrectly determines the remaining full-speed frame time in anticipation of a SOF packet from the host, and erroneously transmits an IN token which results in the later ACK reply being corrupted and replaced by the propagated SOF packet.</p> <p>This type of data corruption is not properly handled by the device state machine, and the device controller must be reset.</p> <p>This sequence has not been seen to occur on commodity USB2.0 hubs, nor on Root Ports that are not provided by a VL805 xHCI controller.</p>
Workarounds	<p>1) VL805 firmware version 0138c1</p> <p>An updated firmware has been pushed to the <code>DEFAULT</code> channel in the <code>raspberrypi-bootloader</code> Apt package on Pi 4 products. This corrects the erroneous hub time calculation. This firmware update is not automatically applied, users must run <code>sudo rpi-eeprom-update -a</code> on the Pi 4 and follow on-screen instructions.</p> <p>2) Linux Kernel xHCI driver patch</p> <p>A kernel update is available for the Raspberry Pi 4-series products that, for VL805 firmware versions earlier than 0138c1, avoids enqueueing single Transfer Descriptors to the controller for affected endpoints during the last microframe of a full-speed frame. This update is available in the <code>raspberrypi-kernel</code> Apt package.</p> <p>2) SDK v1.5.0 / TinyUSB 0.15.0</p> <p>TinyUSB starting at version 0.15.0 adds a workaround for this erratum, and this version is picked up in the v1.5.0 release of the SDK. The <code>dcd_rp2040</code> driver will avoid enabling bulk IN buffers during the last 200µs of a full-speed frame. This reduces available Bulk IN bandwidth by approximately 20%, and selectively enables the Start-of-Frame interrupt.</p> <p>The TinyUSB workaround is not necessary for implementations that will never be connected to a vulnerable VL805 port, for example in a circuit design where RP2040 is directly connected to an on-board hub. The workaround can be disabled by defining <code>TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX=0</code> in your <code>tusb_config.h</code>.</p>

修复于	RP2040B2。RP2040B0和RP2040B1上的软件解决方案。该解决方案未包含于bootrom中的USB大容量存储代码。该软件变通方案要求在USB总线复位期间使用GPIO15。
-----	--

RP2040-E15

参考	RP2040-E15
摘要	若在IN传输过程中发生特定的总线错误，USB设备控制器将会挂起。
描述	<p>若出现以下关键事件序列，USB设备控制器将进入不可恢复状态：</p> <ul style="list-style-type: none"> • RP2040连接至VL805 xHCI控制器，且以全速模式运行。 • 集成集线器检测到下游端口事务转换器流量与上游广播流量（帧起始令牌）之间即将发生线路碰撞。 • 集成集线器在下游进行中的数据包或令牌的PID或CRC部分故意制造比特填充错误。 <p>此序列已知发生于Raspberry Pi 4或Raspberry Pi 400的下游端口，以及数据缓冲区大小超过50字节的Bulk IN端点。在此情况下，集成USB 2.0集线器错误判断主机SOF包的剩余全速帧时间，误发IN令牌，导致后续ACK响应被破坏并被传播的SOF包替代。</p> <p>此类数据损坏未被设备状态机正确处理，设备控制器必须重置。</p> <p>此序列未见发生在普通USB 2.0集线器上，也未见于非VL805 xHCI控制器提供的根端口。</p>
解决方案	<p>1) VL805固件版本 0138c1</p> <p>已向Pi 4系列产品的raspberrypi-bootloaderApt软件包中的DEFAULT频道推送了更新固件。该固件修正了错误的集线器时间计算。此固件更新不会自动应用，用户须在Pi 4设备上运行sudo rpi-eeprom-update -a命令并按照屏幕提示操作。</p> <p>2) Linux内核 xHCI 驱动补丁</p> <p>针对VL805固件版本早于0138c1的Pi 4系列产品，发布了内核更新，避免在全速帧最后微帧期间向受影响端点的控制器排入单一传输描述符。此更新包含于raspberrypi-kernelApt软件包中。</p> <p>2) SDK v1.5.0 / TinyUSB 0.15.0</p> <p>从版本0.15.0开始，TinyUSB引入了解决该错误的规避方案，此版本已纳入SDK v1.5.0发布版。dcd_rp2040驱动程序将在全速帧的最后200μs内避免启用Bulk IN缓冲区。此做法使可用的Bulk IN带宽约减少20%，并有选择性地启用帧起始中断。</p> <p>对于不会连接到受影响VL805端口的实现，例如RP2040直接连接至板载集线器的电路设计，Tiny USB的规避方案并非必需。可通过在tusb_config.h中定义TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX=0来禁用该规避方案。</p>

Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation, Software

RP2040-E16

Reference	RP2040-E16
Summary	Inadequate synchronisation of USB status signals
Description	<p>Within the USB peripheral, certain Host and Device controller events cross from <code>clk_usb</code> to <code>clk_sys</code>. Many of these signals do not have appropriate synchronisation methods to ensure that they are correctly registered when <code>clk_sys</code> is equal to or slower than <code>clk_usb</code>.</p> <p>The following signals lack appropriate synchronisation methods:</p> <p>SIE_STATUS:</p> <ul style="list-style-type: none"> * <code>TRANS_COMPLETE</code> * <code>SETUP_REC</code> * <code>STALL_REC</code> * <code>NAK_REC</code> * <code>RX_SHORT_PACKET</code> * <code>ACK_REQ</code> * <code>DATA_SEQ_ERROR</code> * <code>RX_OVERFLOW</code> <p>INTR:</p> <ul style="list-style-type: none"> * <code>HOST_SOF</code> * <code>ERROR_CRC</code> * <code>ERROR_BIT_STUFF</code> * <code>ERROR_RX_OVERFLOW</code> * <code>ERROR_RX_TIMEOUT</code> * <code>ERROR_DATA_SEQ</code> <p>The bootrom's USB bootloader chains <code>clk_sys</code> from <code>clk_usb</code>, therefore the two clock frequencies are identical and have a fixed phase relationship. In this condition and at extremes of PVT, lab testing has observed that these events may be lost, which results in unreliable USB bootloader behaviour.</p>
Workaround	Run <code>clk_sys</code> faster than <code>clk_usb</code> by at least 10% when the peripheral is in use. Signalling of quasi-static bus states such as reset, suspend, and resume are not affected by this erratum, so <code>clk_sys</code> can be lower in these cases.
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation, software. Not fixed in the RP2040 bootrom.

Watchdog

RP2040-E1

Reference	RP2040-E1
Summary	Watchdog count is decremented twice per tick.
Description	The watchdog (Section 4.7) has a 24-bit counter, that decrements every tick, starting from a user defined value set in <code>LOAD</code> register. There is a logic error which means the counter is decremented twice per tick, instead of once per tick. In a recommended setup where the tick occurs at 1µs intervals, this halves the maximum time between resetting the watchdog counter from ~16.7 seconds to ~8.3 seconds.
Workaround	Use double the desired value in <code>LOAD</code> .
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation, Software

影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

RP2040-E16

参考	RP2040-E16
摘要	USB状态信号同步不足
描述	<p>在USB外设中，某些主控制器与设备控制器事件会从 <code>clk_usb</code> 跨越至 <code>clk_sys</code>。许多此类信号缺乏适当的同步方法，无法确保当 <code>clk_sys</code> 等于或低于 <code>clk_usb</code> 时信号被正确采集。</p> <p>以下信号缺少适当的同步机制：</p> <p><code>SIE_STATUS:</code></p> <p><code>* TRANS_COMPLETE * SETUP_REC * STALL_REC * NAK_REC * RX_SHORT_PACKET * ACK_REQ * DATA_SEQ_ERROR * RX_OVERFLOW</code></p> <p><code>INTR:</code></p> <p><code>* HOST_SOF * ERROR_CRC * ERROR_BIT_STUFF * ERROR_RX_OVERFLOW * ERROR_RX_TIMEOUT * ERROR_DATA_SEQ</code></p> <p>Bootrom的USB引导加载程序将 <code>clk_sys</code> 从 <code>clk_usb</code> 链式传递，因此两者时钟频率一致且具有固定相位关系。在该条件及PVT极限情况下，实验室测试发现这些事件可能丢失，导致USB引导加载程序表现不稳定。</p>
解决方案	当外设使用时， <code>clk_sys</code> 的运行速度至少比 <code>clk_usb</code> 快10%。诸如复位、挂起和恢复等准静态总线状态的信号不受该缺陷影响，因此在这些情况下 <code>clk_sys</code> 可降低。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档，软件。该问题未在RP2040引导只读存储器（bootrom）中修复。

看门狗

RP2040-E1

参考	RP2040-E1
摘要	看门狗计数每个时钟周期减少两次。
描述	看门狗（第4.7节）具有一个24位计数器，从用户在LOAD寄存器中设定的值开始，每个时钟周期递减一次。存在逻辑错误，导致计数器每个时钟周期递减两次，而非一次。在推荐配置下，时钟周期间隔为1μs，该缺陷使看门狗计数器重置的最大间隔时间从约16.7秒减半至约8.3秒。
解决方案	在LOAD寄存器中设置预期值的两倍。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

XIP Flash

RP2040-E8

Reference	RP2040-E8
Summary	Race condition when aborting an XIP DMA stream and immediately starting a new stream
Description	<p>The XIP DMA streaming hardware allows a linear sequences of flash reads to proceed in the background, and be read by the DMA, without subjecting the DMA to the bus stalls caused by a normal XIP-window access. A stream is begun by writing to the STREAM_ADDR register, followed by STREAM_CTR, and can be <i>aborted</i> midway by writing 0 to STREAM_CTR.</p> <p>When a stream is aborted in this way, there is sufficient time for software to load a new address and begin a new stream whilst the final SPI/QSPI access of the aborted stream is still in progress. This causes the newly-loaded stream address to be incremented once before the first data transfer of the new stream sequence, so the entire stream takes place at a 4-byte offset.</p>
Workaround	After clearing STREAM_CTR , immediately perform one dummy read from the uncached XIP window, e.g. <code>(void)*(io_ro_32*)XIP_NOCACHE_NOALLOC_BASE;</code> . If an XIP stream transfer is still in progress, this dummy read will stall until that transfer completes. It is then safe to begin a new stream by writing to STREAM_ADDR followed by STREAM_CTR .
Affects	RP2040B0, RP2040B1, RP2040B2
Fixed by	Documentation, Software

XIP Flash

RP2040-E8

参考	RP2040-E8
摘要	中止XIP DMA流并立即启动新流时存在竞态条件。
描述	<p>XIP DMA 流式硬件允许在后台按线性顺序进行闪存读取，由 DMA 读取，且不会因普通 XIP 窗口访问而引起的总线阻塞影响 DMA。通过先写入 STREAM_ADDR 寄存器，再写入 STREAM_CTR，来启动一个数据流；可通过向 STREAM_CTR 写入 0 来中途停止该数据流。</p> <p>当以此方式中止数据流时，软件有足够时间加载新地址并启动新数据流，而中止数据流的最后一次 SPI/QSPI 访问仍在进行中。这会导致新加载的数据流地址在新数据流序列的第一次数据传输之前递增一次，因此整个数据流以 4 字节偏移进行。</p>
解决方案	清除 STREAM_CTR 后，立即从未缓存的 XIP 窗口执行一次虚拟读取，例如 <code>(void)*(io_ro_32*)XIP_NOCACHE_NOALLOC_BASE;</code> 。如果 XIP 流传输仍在进行中，该虚假读取将阻塞，直到传输完成。然后，先写入 STREAM_ADDR，随后写入 STREAM_CTR，即可安全开始新的流。
影响	RP2040B0, RP2040B1, RP2040B2
修复于	文档、软件

Appendix C: Availability

Raspberry Pi understands the value to customers of long term availability of product and therefore aims to continue supply for as long as practically possible. We expect RP2040 to remain in production until at least January 2041.

Support

For support see the [Pico section of the Raspberry Pi website](#), and post questions on [the Raspberry Pi forum](#).

Ordering code

RP2040 can be ordered in bulk from [Raspberry Pi Direct](#).

Table 638. Part Number

Model	Order Code	Minimal Order Quantity	RRP	Equivalent price per chip
7" reel of 500 × RP2040 chips	SC0914(7)	1+ pcs / Bulk	US\$400.00	US\$0.80
13" reel of 3,400 × RP2040 chips	SC0914(13)	1+ pcs / Bulk	US\$2,380.00	US\$0.70

NOTE

RRP was correct at time of publication and excludes taxes.

附录 C：供应情况

Raspberry Pi 理解客户对产品长期可用性的重视，故致力于在实际可行范围内持续供应。我们预计 RP2040 最少会持续生产至 2041 年 1 月。

支持

有关支持，请参阅 Raspberry Pi 网站的 Pico 部分，并在 Raspberry Pi 论坛发布相关问题。

订购代码

RP2040 可通过 Raspberry Pi Direct 批量订购。

表 638. 零件
编号

型号	订购代码	最低订购 数量	建议零售价	等效每芯片 价格
7" 卷带，含 500 颗 RP2040 芯片	SC0914(7)	1 件以上 / 批量	400.00 美元	0.80 美元
13" 卷带，含 3400 颗 RP2040 芯片	SC0914(13)	1 件以上 / 批量	US\$2,380.00	US\$0.70

① 注意

建议零售价在出版时为正确且未含税款。

Documentation Release History

20 February 2025

- Updated register field types descriptions to use the improved RP2350 wording.
- Add information about running Dual Cortex M0+ processor cores at 200MHz.

15 October 2024

- Corrected minor typos and formatting issues.
- Switched back to separate release histories per PDF.

02 May 2024

- Corrected minor typos and formatting issues.

02 February 2024

- Corrected minor typos and formatting issues.
- Updated ROSC register information.
- Updated to include the new recommended part number for crystals used with RP2040.

14 June 2023

- Corrected minor typos and formatting issues.

03 March 2023

- Corrected minor typos and formatting issues.
- Added errata [E15](#).
- Added package marking specifications.
- Added RP2040 baseline power consumption figures.

01 December 2022

- Corrected minor typos and formatting issues.
- Added RP2040 availability information.
- Added RP2040 storage conditions and thermal characteristics.

文档发行历史

2025年2月20日

- 更新了寄存器字段类型描述，采用改进后的RP2350措辞。
- 新增有关双核Cortex M0+处理器核心以200MHz频率运行的信息。

2024年10月15日

- 修正了轻微拼写错误及格式问题。
- 恢复为每个PDF独立的发布历史记录。

2024年5月2日

- 修正了轻微拼写错误及格式问题。

2024年2月2日

- 修正了轻微拼写错误及格式问题。
- 更新了ROSC寄存器信息。
- 更新内容包括用于RP2040晶体的新推荐零件号。

2023年6月14日

- 修正了轻微拼写错误及格式问题。

2023年3月3日

- 修正了轻微拼写错误及格式问题。
- 新增勘误表E15。
- 新增封装标记规格。
- 新增RP2040基线功耗数据。

2022年12月1日

- 修正了轻微拼写错误及格式问题。
- 新增RP2040供应情况说明。
- 新增RP2040存储条件及热性能资料。

- Replaced SDK library documentation with links to the online version.

30 June 2022

- Corrected minor typos and formatting issues.

17 June 2022

- Corrected minor typos and formatting issues.
- RP2040 now qualified to -40°C, minimum operating temperature changed from -20°C to -40°C.
- Increased PLL min VCO from 400MHz to 750MHz for improved stability across operating conditions.
- Added errata [E12](#), [E13](#) and [E14](#).

04 November 2021

- Corrected minor typos and formatting issues.
- Improved documentation on USB double buffering.
- Updated links to documentation.

03 November 2021

- Corrected minor typos and formatting issues.
- Fixed some register access types and descriptions.
- Added core 1 launch sequence info.
- Described SDK "panic" handling.
- Updated `picotool` documentation.

30 September 2021

- Corrected minor typos and formatting issues.
- Added information about B2 release.
- Updated errata for B2 release.

23 June 2021

- Corrected minor typos and formatting issues.
- Updated information on ADC.
- Added errata [E11](#).

- 将SDK库文档替换为在线版本链接。

2022年6月30日

- 修正了轻微拼写错误及格式问题。

2022年6月17日

- 修正了轻微拼写错误及格式问题。
- RP2040现获-40°C认证，最低工作温度由-20°C调整至-40°C。
- 为提升工作条件下的稳定性，将PLL最小VCO频率自400MHz提高至750MHz。
- 新增勘误 E12、E13 和 E14。

2021年11月4日

- 修正了轻微拼写错误及格式问题。
- 改进了有关 USB 双缓冲的文档内容。
- 更新了文档中的链接。

2021年11月3日

- 修正了轻微拼写错误及格式问题。
- 修正了部分寄存器访问类型及其描述。
- 添加了核心 1 启动序列的相关信息。
- 阐述了 SDK 的 "panic" 处理机制。
- 更新了 `picotool` 工具的文档。

2021年9月30日

- 修正了轻微拼写错误及格式问题。
- 新增了关于 B2 版本发布的说明。
- 更新了 B2 版本的勘误信息。

2021年6月23日

- 修正了轻微拼写错误及格式问题。
- 更新了 ADC 相关信息。
- 新增勘误 E11。

07 June 2021

- Corrected minor typos and formatting issues.
- Added SDK release history.

13 April 2021

- Corrected minor typos and formatting issues.
- Clarified that all source code in the documentation is under the [3-Clause BSD](#) license.

07 April 2021

- Corrected minor typos and formatting issues.
- Added errata [E10](#).

05 March 2021

- Corrected minor typos and formatting issues.
- Improved pinout diagram.

23 February 2021

- Corrected minor typos and formatting issues.
- Changed font.
- Added additional documentation on sink/source limits for RP2040.
- Made major improvements to SWD documentation.
- Added errata [E7](#), [E8](#) and [E9](#).

01 February 2021

- Corrected minor typos and formatting issues.
- Made small improvements to PIO documentation.
- Added missing `TIMER2` and `TIMER3` registers to DMA.

26 January 2021

- Corrected minor typos and formatting issues.
- Added extra information about using DMA with ADC.
- Clarified M0+ and SIO CPUID registers.
- Added more discussion of Timers.

2021年6月7日

- 修正了轻微拼写错误及格式问题。
- 添加了SDK的发布历史记录。

2021年4月13日

- 修正了轻微拼写错误及格式问题。
- 明确规定文档中所有源代码均遵循3-Clause BSD许可证。

2021年4月7日

- 修正了轻微拼写错误及格式问题。
- 新增勘误E10。

2021年3月5日

- 修正了轻微拼写错误及格式问题。
- 优化了引脚排列图。

2021年2月23日

- 修正了轻微拼写错误及格式问题。
- 更换了字体。
- 新增关于RP2040输入/输出极限的附加文档。
- 对SWD文档内容进行了重大改进。
- 新增勘误表E7、E8及E9。

2021年2月1日

- 修正了轻微拼写错误及格式问题。
- 对PIO文档进行小幅改进。
- 在DMA中补充了缺失的TIMER2及TIMER3寄存器。

2021年1月26日

- 修正了轻微拼写错误及格式问题。
- 新增关于在ADC中使用DMA的详细信息。
- 澄清了M0+与SIO CPUID寄存器。
- 增加了关于定时器的详细讨论。

- Renamed books and optimised size of output PDFs.

21 January 2021

- Initial release.

- 重命名书籍并优化了输出PDF文件的大小。

2021年1月21日

- 初始发布。



Raspberry Pi is a trademark of Raspberry Pi Ltd

[Raspberry Pi Ltd](#)



Raspberry Pi is a trademark of Raspberry Pi Ltd

[Raspberry Pi Ltd](#)